TURING 图灵计算机科学丛书

PEARSON Prentice Hall

Java程序设计与问题解决

基础篇（第4版）

Java: An Introduction to Problem Solving and Programming

Fourth Edition

[美] Walter Savitch 著
陈涓 赵振平 译


人民邮电出版社
POSTS & TELECOM PRESS

写给教师的前言

本书是为程序设计及计算机科学方面的第一门课程而撰写的。书中涵盖了一些编程技巧，以及Java程序设计语言的基本概念。适用于短到半学年的课程，也适用于长到一个学年的课程。学生不需要事先具备编程经验，除了一点代数知识外，也不需要更多的数学知识。本书也可以在为那些已经学过其他程序设计课程的学生开设的Java课程中使用，在这种情况下，可以将前面几章指定为学生的课外阅读材料。

本书只使用了Java中的标准类（这些类是Java的一部分），不需要额外的类。

本书所有代码都用Sun公司的Java JDK 5.0 beta 2版测试过。为了保持与本书的全面兼容，你使用的Java必须是5.0版或更高的版本[①]。

本版中的修改

如果你没有用过本书的第3版，就可以跳过本节。如果你用过第3版，本节会告诉你第4版和第3版有哪些不同。

对教师来说，从第3版到第4版的转换是很容易的。在讲授相同的课程时，你可以以相同的顺序讲授基本上相同的主题，只是在涵盖的材料上有极少量的修改。本版最大的修改就是用Java 5.0中提供的新的`Scanner`类取代`SavitchIn`类[②]。如果你想对课程做进一步的修改，本版中还包含了“图形编程补充”小节，你可以选择尽早开始讲述图形编程的内容。本版还添加了对带有类型参数的泛型编程概念的介绍。

第1章~第7章[③]，每章都以一个“图形编程补充”小节作为结束，这个小节涵盖了使用applet以及`JFrame`的图形应用程序和GUI。这些“图形编程补充”小节是可选的。

本版中其他的重要修改都是与升级正文以便同Java 5.0版本相匹配有关的：为键盘输入使用了新的`Scanner`类；解释并使用了自动装箱和拆箱；对向量内容进行升级，以使用泛型类型参数。

最新的Java特性

本版已经进行了更新，使用了Java 5.0的最新特性。特别是，我们为键盘输入使用了新的`Scanner`类，涵盖并使用了自动装箱和拆箱，用类型参数使用向量，还介绍了使用类型

---

① Sun公司修改了它的版本编号方式。版本5.0原来被称为版本1.5，在某些地方你可能会发现它还被称为版本1.5。

② `SavitchIn`类在附录中提供，而且源代码也可以从网上下载。但是，在本书的正文中没有使用`SavitchIn`类。

③ 本书主要包含*Java: An Introduction to Problem Solving and Programming, Fourth Edition*的第1章~第7章的内容，该书的第8章~第14章的内容包含在本书的姊妹篇《Java程序设计与问题解决：高级篇（第4版）》中，附录的内容可从图灵网站（www.turingbook.com）下载。——编者注

参数的泛型编程。编写本书时，我们用Sun公司的Java 5.0 beta 2版检查了所有代码。

灵活性

如果你是一名教师，本书会适应你的教学方法，而不需要你来适应本书。本书没有严格规定所授课程必须涵盖的主题之间的次序，你可以很容易地改变所要讲授的章节间的次序。

本书没有使用专门的库，只使用标准Java库中的标准类。

更早的图形编程

从第1章开始，各章就以可选的“图形编程补充”小节作为章的结尾。这样，从课程一开始，你就可以选择是否讲授图形和GUI编程内容。图形编程补充小节的重点是applet，但也涵盖了用`JFrame`类构建的GUI。希望推迟讲授图形编程内容的教师可以推迟讲授或跳过这些“图形编程补充”小节。

问题求解及编程技术的内容

本书以教学生基本的问题求解方法及编程技术为设计目标，而不仅仅是一本有关Java语法的书。书中包含了大量的案例学习和编程提示，以及很多讲解了重要问题求解方法及编程技术的小节，如循环设计技术、调试技术、编程风格、抽象数据类型、基本的面向对象编程技术（包括UML和事件驱动编程）以及使用类型参数的泛型编程。

面向对象技术与传统技术

任何实际讲授Java的课程都必须很早就讲授类的概念，因为Java中所有的概念都涉及了类。一个Java程序是一个类，字符串数据类型是一个类，甚至等于运算符（==）的行为都取决于它是在与类中的对象进行比较，还是在与一个较简单的数据进行比较。除非利用那些极长且复杂的“魔法公式”，否则，类是无法避免的。本书很早就引入了类的概念。在第1章和第2章介绍了一些类的使用，第4章介绍了如何定义类。所有关于类的基本信息，包括继承在内，都在第7章结束之前进行了交待（即使略过第6章也是如此）。但是，可以将某些与类有关的主题，包括继承，推迟到课程的后期或高级课程中介绍。

尽管本书很早就引入了类的概念，但它并没有忽略自顶向下的设计方法和循环设计技术这样的传统编程技术。这些较老的主题可能不再那么吸引人，但仍然是所有初学者必须掌握的内容。

语言细节及示例代码

本书讲授的是编程技术，而不仅仅是Java语言。但是，对于一门不讲授编程语言的入门性编程课程，无论是学生还是教师都不会感到满意的。在你帮助学生克服对语言细节的恐惧之前，通常是无法将他们的注意力集中到更大的问题上的。因此，本书给出了对Java语言

特性的完整解释，以及大量的示例代码。书中提供了完整的程序，以及作为示例的输入和输出。在很多情况下，除了正文中那些完整的示例之外，还可以从因特网上获得一些额外的完整示例。

自测题

本书每章都配有自测题。这些问题的难度级别跨度很大。有些只需要用一句话来回答，而有些则需要编写一个完整的、重要的程序。在每章的结尾都给出了所有自测题的完整答案，包括那些需要编写完整程序的题目。

课堂检验

书中所有资料都经过完整的课堂检验，并根据检验结果对很多资料和表达方式进行了修改。

配套资料

你可以从出版商或者因特网获取以下的配套资料。

配套资源

在本书的配套网站http://www.prenhall.com/savitch上提供了本书的源代码、更多编程示例以及下载Java编译器和编程环境的链接。

教师资源指南

教师工具中包含各章的教师资源指南，指南中包含了大量的授课提示、带有答案的测试题、很多编程练习的答案、PowerPoint幻灯片以及其他一些授课资源。教师可以与Prentice Hall销售代表联系，以获取访问教师网站的相关信息。可以通过网址http://www.prenhall.com与Prentice Hall联系，获取销售代表的名字和编号。

Walter Savitch
wsavitch@ucsd.edu
http://www.cse.ucsd.edu/users/savitch

写给学生的前言

本书的目的是讲授Java程序设计语言，但更重要的是，讲授基本的编程技术。本书不要求事先具备编程经验，除了一些简单的代数知识之外，也不需要更多的数学知识。但是，为了最大程度地从本书获益，在你的机器上应该装有Java，这样你就可以练习书中给出的示例和技巧了。你应该安装Java 5.0版本（或更高的版本）。（Sun公司修改了它的版本编号方式。版本5.0原来被称为版本1.5，在某些地方你可能会发现它仍然被称为版本1.5。如果你有一个名为“版本1.5”的Java的副本，也应该可以。）

如果以前编写过程序

使用本书不要求有编程经验。本书是为初学者设计的。如果你碰巧有过使用其他编程语言的经验，不要假设Java和你习惯使用的编程语言是一样的。所有的语言都是不同的，它们之间的区别即使很小，也足以带来一些问题。你至少应该读一下1.3节的内容和第2章中所有内容。当阅读到第4章的时候，阅读整章是比较明智的。

如果你以前曾经用C或C++编写过程序，那么，向Java的转换可能会比较麻烦。乍一看，Java好像和C或C++基本上是一样的。但是，Java和这些语言有很大的不同，你必须认识到这些区别。附录K对Java和C++进行了比较，以帮助你了解它们之间的区别。

正文中的程序代码

本书中所有的程序和其他软件示例都可以从本书的配套网站上下载，这样，你不用将这些示例输入计算机，就可以用它们进行练习了。

获取Java环境

在本书的配套网站上有可以下载Java编译器和编程环境的链接。对初学者来说，我们推荐使用Sun公司的Java SDK作为Java编译器及相关软件，推荐TextPad作为简单的编辑环境来编写Java代码。下载Java SDK时，一定要下载版本号为5.0的版本或更新的版本。

本书的配套网站

在本书的配套网站http://www.prenhall.com/savitch上提供了本书中的源代码、额外的编程示例及下载Java编译器和编程环境的链接。

自测题

每章中都配有大量的自测题。在每章的结尾都给出了这些问题的完整答案。要对所学

的知识进行练习，最好的方法就是在看答案之前做这些自测题。

本书也是一本参考书

本书除了可以作为教材，还可以并且应该将它当作一本参考书。“快速参考”中会有一个很短的条目，给出了所有与那个主题有关的基本内容。可以用这种方法来查看Java语言及编程技术方面的细节。

每章的“小结”都对该章的主要内容进行了简要的总结。你可以根据这些小结中的内容对该章进行复习，或者根据这些内容来查看Java语言的一些细节。

我们期待读者的意见

本书是为你编写的，我们期望听到你对本书的任何评论。你可以用电子邮件(wsavitch @ucsd.edu）与我联系。

遗憾的是，我不能为你提供编程练习的答案。只有采用本书作为教材的教师才可以从出版商那里获取部分答案。要想获取编程练习方面的帮助，你只能与你的老师联系。但是，每章的结尾都有所有自测题的答案。

Walter Savitch
http://www.cse.ucsd.edu/users/savitch

致　谢

我要感谢我所在的加州大学圣迭戈分校的计算机科学与工程系，本书很多素材在这里得到了检验。在我授课的班级中，很多学生都帮助我对本书的初稿进行了校对。这些学生的建议和在课堂上试用本书的教师的建议对形成本书的最终版本有着极大的帮助。我要特别感谢加州大学萨克拉门托分校的Carole McNamee和加州大学圣迭戈分校的Paul Kube，他们对本书的早期版本提出了详细的反馈意见，并在课堂进行试用。我还要特别感谢杨百翰大学的Robert Burton，他为本版的草稿准备了详细的分析材料。他们的建议对本书的最终成型提供了巨大的帮助。

我要感谢所有花时间阅读本书早期版本书稿的审阅者。他们提供了非常宝贵的、详细的评论和建议，这个新的版本也继续从这些评论和建议中受益。按照字母顺序，这些审阅者是：

Jim Buffenbarger——爱达荷州立大学（Idaho State University）

Robert P. Burton——杨百翰大学（Brigham Young University）

Steve Cater——凯特林大学（Kettering University）

Martin Chelten——穆尔帕克社区学院（Moorpark Community College）

Michael Clancy——加州大学伯克利分校（University of California, Berkeley）

Tom Cortina——纽约州立大学石溪分校（SUNY, Stony Brook）

Prasun Dewan——北卡罗来纳大学（University of North Carolina）

Laird Dotnan——Sun公司（Sun Microsystems, Inc.）

H.E. Dunsmore——普度大学拉斐特分校（Purdue University, Lafayette）

Adel Elmaghraby——路易斯维尔大学（University of Louisville）

Gobi Gopinath——萨福克县社区学院（Suffolk County Community College）

Le Gruenwald——俄克拉何马大学（University of Oklahoma）

Gopal Gupta——得克萨斯大学达拉斯分校（University of Texas, Dallas）

Ricci Heishman ——北弗吉尼亚社区学院（North Virginia Community College）

Robert Herrmann——Sun公司（Sun Microsystems, Inc., Java Soft）

Robert Holloway——威斯康星大学麦迪逊分校（University of Wisconsin, Madison）

Lily Hou——卡内基-梅隆大学（Carnegie Mellon University）

Rob Kelly——纽约州立大学石溪分校（SUNY, Stony Brook）

Michele Kleckner——埃隆学院（Elon College）

Mike Litman——西伊利诺伊大学（Western Illinois University）

Blayne Mayfield——俄克拉何马州立大学（Oklahoma State University）
John Motil——加利福尼亚州立大学北岭分校（California State University, Northridge）
Michael Olan——斯托克顿州立大学（Stockton State）
James Roberts——卡内基-梅隆大学（Carnegie Mellon University）
Alan Saleski——罗耀拉大学芝加哥分校（Loyola University, Chicago）
Nan C. Schaller——罗切斯特理工学院（Rochester Institute of Technology）
Ryan Shoemaker——Sun公司（Sun Microsystems, Inc.）
Ken Slonneger——艾奥瓦大学（University of Iowa）
Donald E. Smith——罗杰斯大学（Rutgers University）
Boyd Trolinger——巴特学院（Butte College）
Subramanian Vijayarangam——麻省大学洛厄尔分校（University of Massachusetts, Lowell）

我还要感谢本版的审阅者们：

Robert Burton——杨百翰大学（Brigham Young University）
Ed Gellenbeck——中华盛顿大学（Central Washington University）
Anthony Larrain——德保大学（Depaul University）
Michael Long——加州大学（California State University）
Drew McDermott——耶鲁大学（Yale University）
Ken Slonneger——艾奥瓦大学（University of Iowa）
Navabi Tadayon——亚利桑那州立大学（Arizona State University）
Richard Whitehouse——亚利桑那州立大学（Arizona State University）
Michael Young——俄勒冈大学（University of Oregon）

我还要感谢Prentice Hall所有负责组织本书审阅和出版工作的人员，我尤其要感谢Toni Holm、Patrick Lindner和Irwin Zucker。我要特别感谢我的出版人Alan Apt，感谢他在本书的编写和出版过程中提供的非常宝贵的支持和建议。这些优秀的人都做得很棒！

Lew Rakocy准备了教师指南的编程答案，Robert Burton准备了PowerPoint幻灯片。我要感谢他们细心的工作。

我要感谢Sun公司允许我在很多GUI例子中使用Duke图标。

Walter Savitch

目　录

第1章

计算机与Java概述

想造一台可以解决那些确实非常难的数学问题的机器绝不是毫无可能的。但你必须一步一步来。我认为电是最可依赖的。

——查尔斯·桑德斯·皮尔士，美国哲学家（1839—1914）

本章对计算机的硬件和软件进行了简单的概述，介绍的大部分内容都适用于任何一种语言的编程，而不是只适用于Java编程。在对软件的讨论中包含了对面向对象编程这种程序设计方法的描述。1.3节介绍了Java语言，并对一个Java程序示例进行了解释。

本书各章都以“图形编程补充”作为结尾，作为第一个“图形编程补充”小节，1.4节最先对Java语言的图形编程能力进行了前期介绍。这些图形编程补充小节是选读的，但如果想阅读这些小节的内容，就应该从1.4节开始。

目标

- 概述计算机硬件和软件。
- 总体介绍程序设计的基本技巧，并专门介绍面向对象编程的技术。
- 综述Java编程语言。
- 选读，介绍applet和一些图形编程基础知识。

预备知识

本章并不需要你以前有任何编程经验，但假定你有一台计算机可用。为了最大程度地从本章及本书的其余部分获益，你应该有一台安装了Java语言的计算机，这样就可以实践所学的内容了。前言中介绍了获取免费Java语言副本的方式。

在学习第2章之前，至少要阅读1.3节的内容。可以在阅读1.1节和1.2节之前阅读1.3节和1.4节（实际上，可以以任意的顺序阅读1.1节~1.3节）。但阅读1.4节之前必须阅读1.3节。

1.1 计算机基础

分析机并不能发明任何新的东西。它可以完成任何我们知道如何命令它完成的任务。它可以进行分析，但没有预测任何解析关系或真理的能力。它的作用就是帮助我们利用一些我们已经很熟悉的东西。

——艾达·奥古斯塔，拉夫拉斯伯爵夫人，世界第一位程序员（1815—1852）

计算机系统由**硬件**（hardware）和**软件**（software）组成。硬件是物理机器。计算机的一

套指令被称为一个**程序**（program）。所有用来向计算机发送指令的不同类型的程序称为软件。本书中要讨论的是软件，但为了更好地理解软件，了解一些计算机硬件的基本知识还是很有帮助的。

1.1.1　硬件与存储器

如今使用的大部分计算机都有相同的基本组件，并用基本相同的方法进行配置。它们都有键盘和鼠标这样的输入设备以及显示器和打印机这样的输出设备，它们还会有两三种其他基本组件，这些组件通常都放在机箱里面，这样我们就不太容易看见它们。这些组件包括处理器与主存储器和辅助存储器两种存储器。

处理器（processor）是计算机内部的一种设备，它执行程序的指令。（处理器也被称为CPU，即**中央处理器**（central processing unit）。）如果要买一台PC，会被告知这台计算机使用的是什么芯片。这个**芯片**指的就是处理器。当今，奔腾处理器是比较知名的芯片之一。处理器会执行程序中的指令，但它只能执行一些非常简单的指令，例如将数字或其他项从存储器中的一个位置移到另一个位置上去，或者执行一些加、减法这样的简单算术运算。计算机的能力来自其速度和程序复杂性。从概念上来讲，硬件的基本设计是很简单的。

计算机的**存储器**（memory）中装有计算机要处理的数据以及计算机的中间计算结果。计算机有两种基本的存储器类型：主存储器和辅助存储器。所有与计算机一起使用的各类磁盘驱动器、软盘和光盘都是**辅助存储器**（auxiliary memory），它们基本上都是永久存储器。（辅助存储器也被称为**二级存储器**（secondary memory）。）用来存储中间计算结果（以及当前正在运行的程序）的工作存储器称为**主存储器**（main memory）。编写程序时最需要了解的是主存储器的特点。主存储器中包含当前程序及其操作的大部分数据。

为了更具体地说明这个问题，来看一个例子。比如，你可能会听人们说一台台式机（PC）有256MB的RAM和50GB的硬盘驱动器（或者其他容量的RAM和硬盘驱动器）。RAM（Random Access Memory，随机存储器）是主存储器，硬盘驱动器是主要的（但不是唯一的）辅助存储器。字节（Byte, B）是存储器的基本单位。因此，256MB的RAM大约是2亿5千6百万字节的存储器，50GB的硬盘驱动器大约是500亿字节的存储器。那么，一个字节到底表示什么呢？

位（bit）是一个只能取0和1两个值的数字（实际上，可以取任意两个值，但这两个值通常被写成0和1）。一个**字节**（byte）等于存储器中的8个位，也就是可以存储8个位的存储器基本单元，每个位都可以是0或1。主存储器和辅助存储器都以字节为单位来度量。在主存储器中，字节的组织是非常重要的。计算机的主存储器由一长串编了号的单元组成，每个单元都可以存储一个字节的信息。字节的编号称为**地址**（address）。可以将一份数据，比如数字或键盘字符，存储在这样的一个字节中。稍后，当计算机要恢复这些数据时，可以根据字节的地址找到这些数据项。

可以将各种类型的数据，比如字母、数字和字符串，编码成一系列的0和1，并将其存放在计算机存储器中。一个字节正好可以存储一个键盘字符。这就是将计算机存储器划分成8位字节，而不是划分成其他位大小的原因之一。但是，要存储大的数字或一串字母，计算机就需要多个字节。当计算机需要存储一份无法放入单字节中的数据时，会使用许多相邻的字节。然后，这些相邻的字节会被看成单个的、较大的**存储单元**（memory location），并将第一个字

节的地址作为整个较大的存储单元的地址。图1-1显示了一个典型的计算机主存储器是如何划分存储单元的。这些单元之间的边界并不是由硬件固定的。运行不同程序时，单元的大小和边界的位置可能会有所不同。

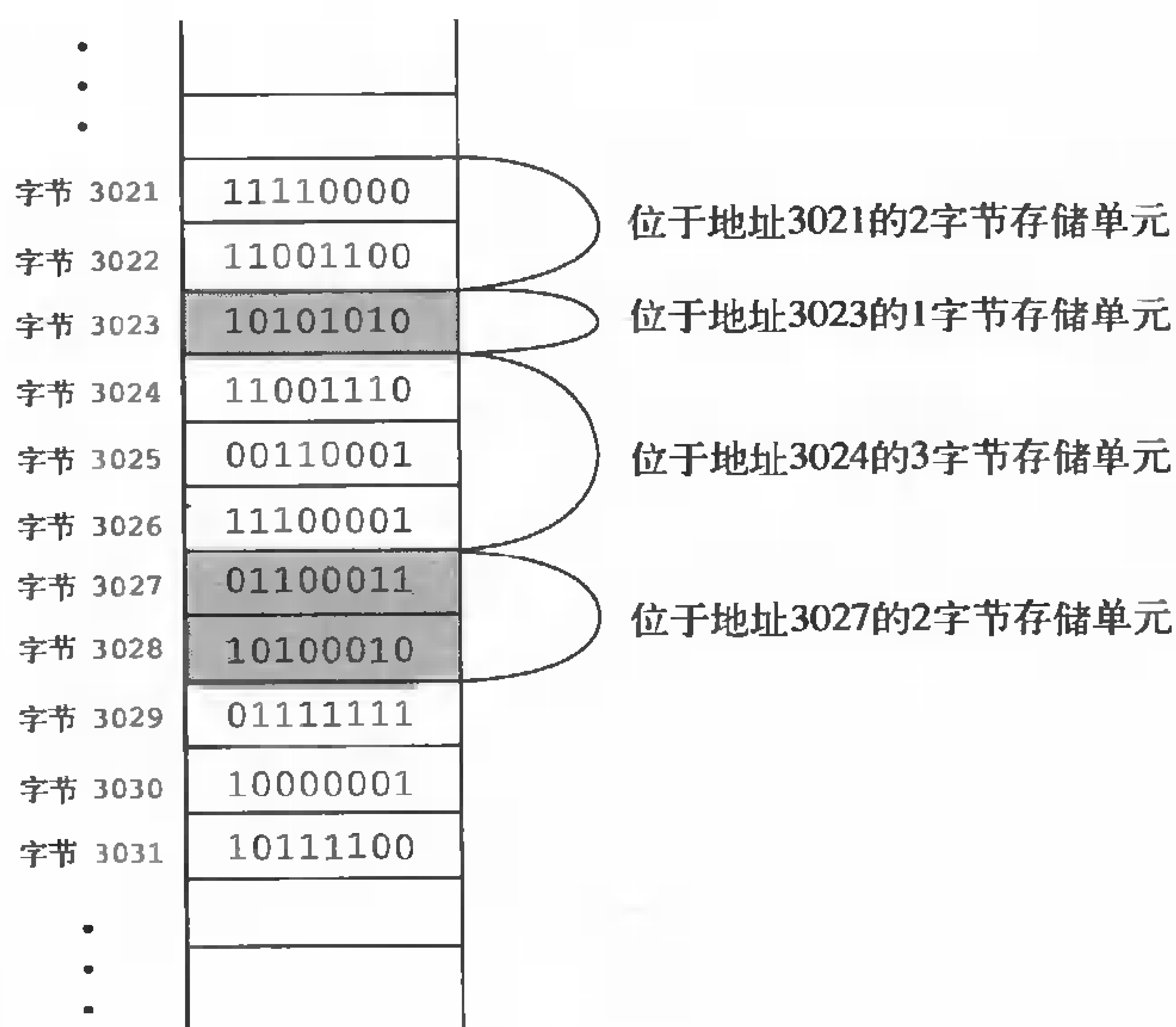


图1-1 主存储器

回想一下，只有当计算机运行程序时，才会用到主存储器。辅助存储器基本上是以一种永久模式来存储数据的。辅助存储器也被划分为字节，但这些字节又被组织成了名为**文件**(file）的更大单元。文件可以（以编码形式）包含几乎所有类型的数据，比如一段程序、一个字母、一串数字或一幅图片。文件的重要特性是：它有名字，而且可以装载数据。编写Java程序时，要将程序存储在一个文件中。文件存储在辅助存储器中（通常是某种磁盘存储器)，要运行程序时，将程序从辅助存储器复制到主存储器中。

通常将文件组织成**目录**（directory）或**文件夹**（folder）这样的文件组。文件夹和目录是同一个事物的两个名字。有些计算机系统使用文件夹，有些计算机系统使用目录。

常见问题：为什么是0和1？

计算机使用0和1是因为只有两个稳定状态的物理设备比较容易制造。但是，在编程的时候，通常不需要关心数据是编码为0还是1。编程时可以看成计算机是将数字、字母或字符串直接存储到存储器中去的。

数字0和1没有什么特别之处。我们用任意两个名字，比如A和B，或者真和假，来取代0和1也是一样的。重要的是底层的物理设备有两个稳定的状态，比如开和关，或者高电压和低电压。把这两个状态称为0和1只是一种约定，不过基本上大家都遵守这个约定。

快速参考：字节与存储单元

字节是可以存储8个数字的存储单元，每个数字只能是0或1。计算机的主存储器被划分成编了号

的字节。字节的编号被称为它的**地址**。要存储的数据太大以至于放不进单个字节中时，计算机会使用许多相邻的字节。这些相邻的字节被当作一个较大的存储单元，并将第一个字节的地址作为整个较大存储单元的地址。

1.1.2 程序

可能你对程序是什么已经有些概念了，实际上，你一直都在使用程序。比如，文本编辑器和字处理器（比如Word）就是程序。银行的ATM机实际上是一台正在运行一个程序的计算机。**程序**只是计算机要遵循的一系列指令。

图1-2表明可以用两种方式来观察一个程序的运行。采用第一种方式时，要忽略图中的虚线框。剩下的就是运行程序时实际发生的事情了。注意，在运行程序时，计算机的输入通常有两种类型。程序是一种输入，它包含了计算机需要执行的指令。另一种输入通常被称为程序的**数据**，是计算机程序要处理的信息。例如，如果程序是一个简单的拼写检查程序，数据就是需要检查的文本。就计算机而言，数据和程序本身都是输入。输出就是计算机根据程序的指令运行产生的结果。如果程序对某些文本的拼写进行了检查，输出可能就是拼错了的单词列表。当你交给计算机一个程序和一些数据，并告诉计算机遵循程序中的指令操作时，就是在根据这些数据**运行**（running）程序，而计算机就是在根据这些数据**执行**（execute）程序。

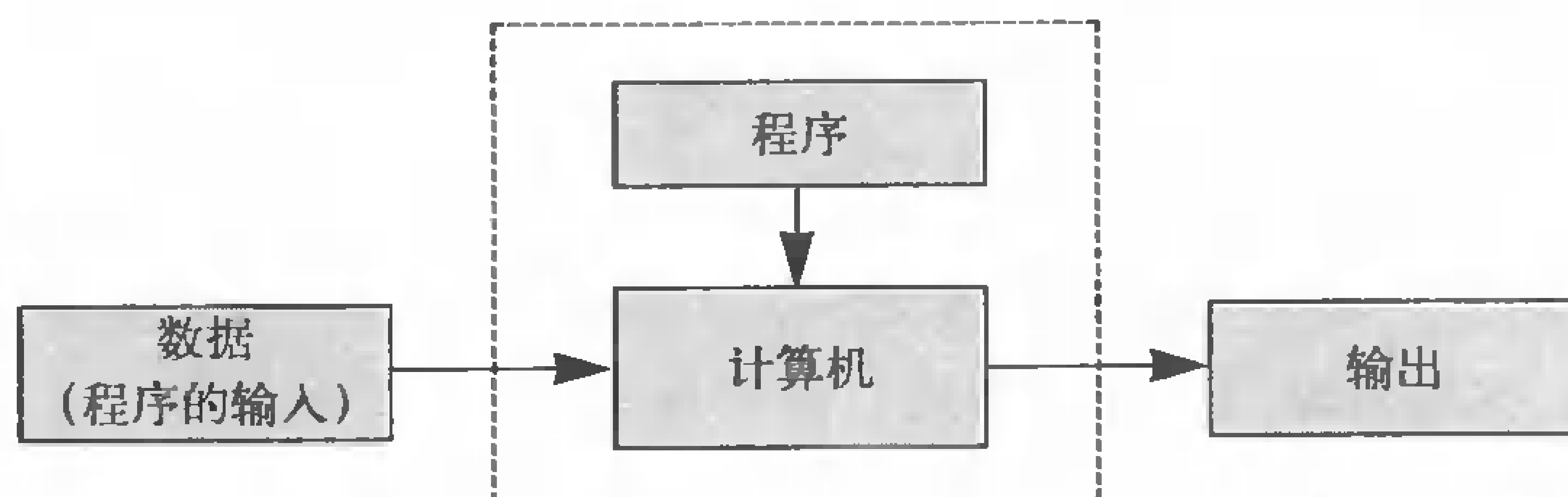


图1-2　运行一个程序

第一种方式查看的是运行程序时实际发生的情况，但这并不总是我们所考虑的程序运行的方式。另一种方式是把数据当作程序的输入。在第二种方式中，计算机和程序被看成一个单元，将数据作为输入，并产生输出。在这种方式中，用虚线框表示结合在一起的程序-计算机单元。采用这种方式时，将数据看成是程序的输入，输出则来自程序的输出。尽管我们都知道计算机是存在的，但这里假设它只是用来辅助程序运行的。在设计程序时，程序员会觉得第二种方式更有用一些。

计算机中安装的程序比你想象的要多。大部分通常认为是“计算机”的东西实际上都是程序而不是硬件。你第一次打开计算机时，就已经在运行一个程序，并与其进行交互了。这个程序被称为**操作系统**（operating system）。操作系统是一种监控程序，用来监控计算机的所有操作。如果想运行一个程序，就要告诉操作系统你想做什么。然后，操作系统会检索到那个程序，并启动它。运行的程序可能是一个文本编辑器、一个用来在万维网（World Wide Web）上冲浪的程序或者用Java语言编写的某个程序。你可以通过用鼠标点击图标、选择菜单项或者输入一条简单的命令来通知操作系统运行程序。由此可见，可能会被当成“计算机”的东西实际上是操作系统。常见的操作系统有DOS、微软公司的Windows、苹果公司的

(Macintosh) Mac OS、Linux和UNIX。

常见问题：到底什么是软件？

软件这个词简单地说就是指程序。因此，一个软件公司就是一个生产程序的公司。计算机上的软件就是计算机中程序的集合。

1.1.3 编程语言与编译器

大部分现代编程语言都设计得相对易于编写和理解。这些为了供人们使用而设计的编程语言被称为**高级语言**（high-level language）。Java是一种高级语言。大部分常用的编程语言，如Pascal、FORTRAN、C、C++、BASIC和Visual Basic等也都是高级语言。遗憾的是，计算机硬件无法理解高级语言。在运行一个用高级语言编写的程序之前，必须将其翻译成一种计算机可以理解的语言。计算机可以（更直接地）理解的语言被称为**低级语言**（low-level language)。将程序从Java这样的高级语言翻译成低级语言的过程完全或部分地由一个名为**编译器**（compiler）的程序实现。

运行一个用Java这样的高级语言编写的程序时，实际上运行的是该程序的低级语言翻译版本。因此，在运行高级语言程序之前，首先必须对该程序运行编译器。做这项工作时，就是在**编译**程序。

编译器产生的低级语言通常称为**机器语言**（machine language）或**汇编语言**（assembly language)。计算机可以直接理解的语言被称为机器语言。汇编语言和机器语言基本上是一回事，但在计算机上运行汇编语言之前，还要对它进行少量额外的翻译。通常，这种额外翻译是自动完成的，不需要你来操心。实际上看起来就像是在运行由编译器产生的程序一样。

刚才所说的高级语言翻译过程的缺点之一是，对大多数编程语言来说，要为每种类型的计算机和每种操作系统使用不同的编译器。如果想在3种不同类型的计算机上运行你的高级语言程序，就需要使用3种不同的编译器，将程序编译3次。而且，如果制造商推出一种新型的计算机，就需要一组程序员为这种计算机编写一个新的编译器。由于编译器是很大的程序，编译器的开发非常昂贵，而且很耗时，因此这就成了一个问题。尽管要付出这样的代价，大部分高级语言编译器还是以这种方式工作。但是，Java使用了一种稍有不同但更通用的编译方式，我们将在1.1.4节对其进行讨论。

使用编译器时，输入编译器程序的以及从编译器程序中输出的都是程序，所以，术语可能会有些混乱。满眼都是这样或那样的程序。为了避免混淆，我们将输入程序（在我们的例子中会是一个Java程序）称为**源程序**（source program）或**源代码**（source code)。编译器产生的、翻译得到的低级语言程序通常称为**目标程序**（object program）或**目标代码**（object code)。**代码**（code）一词只是用来表示一个程序或程序的一部分。

快速参考：编译器

编译器是一个程序，它负责将Java程序这样的高级语言程序翻译成计算机基本上可以直接理解的、更简单的语言编写的程序。

1.1.4 Java字节码

Java编译器不会将程序翻译成特定计算机的机器语言。相反，它会将Java程序翻译成一种被称为**字节码**（byte-code）的语言。字节码不是任何特定计算机的机器语言。字节码是一台假想计算机的机器语言，这台假想计算机就像是所有计算机的混合体一样。这台假想计算机被称为**Java虚拟机**（Java Virtual Machine）。Java虚拟机与任意一台特定的计算机都不完全相同，但它与所有典型的计算机都类似。因此，将用字节码编写的程序翻译成任意特定计算机的机器语言都是很容易的。实现这种翻译的程序被称为**解释器**（interpreter）。解释器将每条字节码指令翻译成你所用计算机的机器语言所表示的指令，然后在计算机上执行这些指令。因此，解释器对字节码指令的翻译和执行是逐条进行的，而不是一次翻译整个字节码程序。但实际上你唯一需要了解的细节就是：解释器可以通过某种方式使你的计算机运行Java字节码。负责翻译及运行Java字节码的解释器也被称为Java虚拟机（因为它是底层假想计算机的一种实现，这个假想计算机被称为Java虚拟机，字节码就是基于这种假想计算机的）。

为了在计算机上运行Java程序，要进行下列工作：首先，用编译器将Java程序翻译成字节码。然后，使用你所用机器的字节码解释器将每条字节码指令翻译成机器语言，并运行这些机器语言指令。图1-3示出了整个过程。

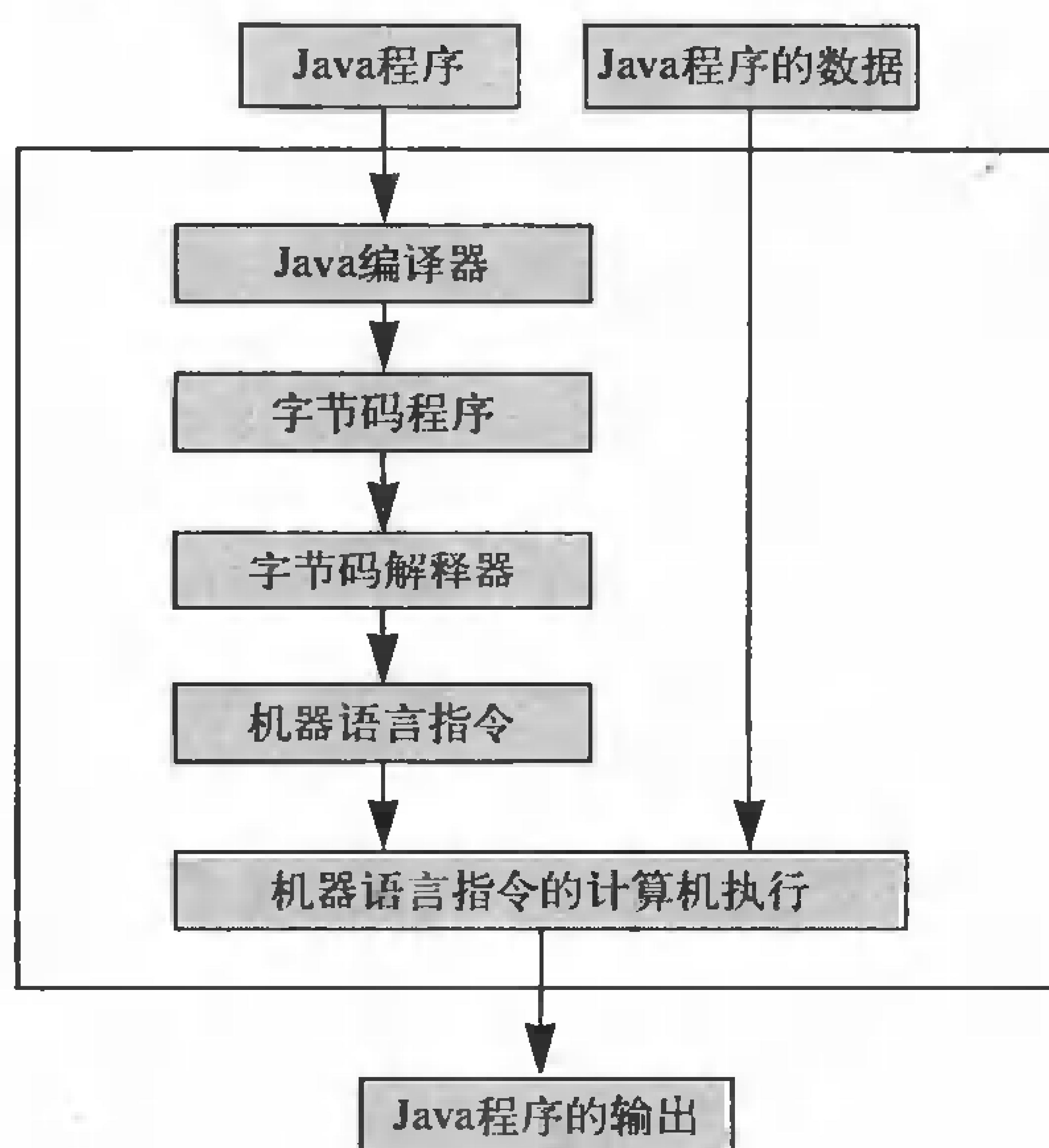


图1-3　编译并运行Java程序

听起来好像Java字节码只是在该过程中额外添加了一步。为什么不编写直接将Java翻译成特定计算机所用机器语言的编译器呢？这是可以做到的，大多数其他的编程语言也是这么做的。而且，那种技术生成的机器语言程序通常运行得更快些。但是，Java字节码赋予了Java一项很重要的优点，即可移植性。将Java程序编译成字节码之后，就可以在任意一台计算机上使用那些字节码了。在另一种类型的计算机上运行你的程序时，不需要对其重新编译。这就意味着你可以通过因特网将字节码发送给另一台计算机，并且可以很容易地在那台计算机上运行你的程序。这就是Java适用于因特网应用的原因之一。

可移植性还有其他一些优点。当制造商开发出一种新型计算机时，Java的创建者们就不需要再设计新的Java编译器了。一种Java编译器可以适用于所有的计算机。这就意味着可以很快并且很经济地将Java添加到新型计算机中去。当然，为了将字节码指令翻译成特定计算机的机器语言指令，每种类型的计算机都必须有自己的字节码解释器，但与编译器相比，这些解释器都是很简单的程序。

快速参考：字节码

Java编译器将Java程序翻译成一种名为**字节码**的语言。这种字节码不是任何特定计算机的机器语言，但它与大多数通用计算机的机器语言类似，并且很容易将其翻译成任意特定计算机的机器语言。每种类型的计算机都有自己的翻译器（称为解释器），负责将字节码指令翻译成特定计算机的机器语言指令。

虽然了解Java字节码是很重要的，但在日常的编程工作中，你甚至都不需要知道程序字节码的存在。通常，只需要给出两条命令，一条用来编译程序（将其编译成字节码），另一条用来运行程序。**运行命令**（run command）会对字节码执行Java字节码解释器。这条运行命令可能会被叫作“run”或其他名字，但不太可能叫作“interpret”。可以认为运行命令是在运行编译器产生的任何内容，甚至都不用考虑它实际运行的是字节码而不是机器语言。

常见问题：为什么称其为“字节码”？

由字节码和机器语言代码这样的低级语言编写的程序由指令组成，每条指令都可以存放在内存的几个字节中。这也许就是字节码有这样一个名字的原因。对Java的设计者来说，字节码看起来一定很像“一群字节”。

1.1.5 类装载器

编写Java程序时，很少将所有代码段都写在一个文件里。相反，它通常是由称为**类**（class）的不同代码段组成的。这些类通常由不同的人编写，而且每个类都是分别编译的。这样，每个类（每个代码段）就会被翻译成不同的字节码片段。要运行程序，就要将这些不同类的字节码连接起来。连接是由一个名为**类装载器**（class loader）的程序实现的。这种连接通常都是自动完成的，因此一般不必去关心它。在其他编程语言中，与Java的类装载器相对应的程序被称为*连接器*（linker）。

自测题

自测题的答案在每章的最后。

1. 计算机中有哪两种类型的存储器？
2. 什么是软件？
3. 用来计算两个数字之和的程序需要哪些数据？
4. 用来计算在一门课程中所参加的所有测验的平均分的程序需要哪些数据？
5. 机器语言程序、高级语言程序和用Java字节码表示的程序之间有什么区别？
6. Java是高级语言还是低级语言？

7. Java字节码是高级语言还是低级语言？
8. 什么是编译器？
9. 什么是源程序？
10. 将Java字节码指令翻译成机器语言指令的程序叫什么名字？

1.2 程序设计

"到时候了，"海象说，"咱们来东拉西扯：谈谈靴子、船舶和封蜡；还有白菜和国王……"

——刘易斯·卡洛尔，《爱丽丝镜中奇遇记》

编程是一种创造性的过程。我们无法确切地告诉你如何编写一个程序来完成你想让它完成的任务，但是可以给出一些有经验的程序员认为非常有用的技巧。本节对其中的一些技巧进行了讨论，这些技巧不仅可用于Java编程，而且适用于几乎所有的编程语言。

1.2.1 面向对象编程

Java是一种**面向对象编程**（object-oriented programming，OOP）语言。什么是OOP呢？我们周围的世界是由对象组成的，比如人、汽车、建筑物、树木、靴子、船舶、蜂蜡、白菜、国王等。每种对象都可以执行某些动作，每种动作都会对世界上另外一些对象产生某种影响。OOP是一种编程方法，它将程序也看成是由对象组成的，这些对象之间通过动作进行交互。

如果程序模拟的是现实世界中的事物，这种面向对象的方式就更容易理解一些。例如，考虑一个仿真进出高速公路的主体交叉道以分析交通流量的程序。程序中需要一个对象来模拟进入交叉道的每辆汽车，可能还需要其他对象来仿真高速公路的每条车道，等等。

面向对象编程有自己术语。对象的称谓很恰当。对象可以采取的动作称为**方法**（method）。称同一种类型的对象具有相同的*类型*（type），或者，更经常地称其属于同一个**类**（class）。例如，在一个仿真程序中，所有的仿真汽车可能都属于同一个类，这个类很可能称为`Automobile`类。类中所有的对象都拥有相同的方法。这样，在一个仿真程序中，所有的汽车都有相同的方法（或可能的动作），比如向前开、向后开、加速等。这并不意味着所有的仿真汽车都完全相同。它们可以有不同的属性，程序将数据（也就是一些信息）与每个特定的汽车对象关联起来，以说明对象的不同属性。例如，与汽车对象相关的数据可能包括一个描述汽车型号的单词和一个说明其当前速度的数字。（亲自用Java编程语言定义类时，所有这些问题就会更加明晰了。）

快速参考：对象、方法和类

对象是一种程序构造，有与之相关的数据（即信息），并可以执行某些动作。程序运行时，对象之间会进行交互，以实现程序设计需要完成的任务。对象执行的动作称为**方法**。**类**是一种或一类对象。同一个类的所有对象都拥有相同类型的数据和相同的方法。

正如你将会看到的那样，同样的面向对象方法适用于各种类型的计算机程序，而不仅仅局限于仿真程序。面向对象编程不是一种新方法，但直到20世纪90年代早期，它在仿真程序之外的应用仍不是很广泛。

面向对象编程使用类和对象，但并不只是用老方法来使用它们。它在使用的同时遵循一些设计原则。下面就是面向对象编程中3个主要的设计原则：

- 封装；
- 多态性；
- 继承。

本章将简单介绍每个原则，并且稍后在适当的地方对其进行更完整的介绍。

快速参考：面向对象编程

面向对象编程是一种编程方法，它将程序看成对象的集合，这些对象通过称为**方法**的动作进行交互。面向对象编程在遵循某些设计原则的基础上使用了对象，其主要设计原则包括封装、多态性和继承。

1.2.2 封装

封装（encapsulation）听起来好像就是把东西放到一个小盒子里，或者换句话说，就是把东西包装起来。到目前为止，这种直觉是正确的。但是，封装中最重要的部分并不是简单地将东西放到一个小盒子里面，而是表示小盒子里的内容只有部分是可见的。下面来看一个例子。

假设你想驾驶一辆汽车。对汽车最重要的描述是什么呢？很明显不是描述汽缸数，也不是描述汽缸如何通过下面这个循环：吸入空气和汽油，点燃油气混合物，以及排出废气。这些细节不会帮助你了解如何驾驶一辆汽车。

对一个想学习驾驶的人来说，对汽车最有用的描述是由下面这类信息组成的：

- 如果你的脚踩在油门踏板上，汽车就会开得更快一些。
- 如果你的脚踩在刹车踏板上，汽车就会减速并最终停下来。
- 如果将方向盘向右转，汽车就会向右转。
- 如果将方向盘向左转，汽车就会向左转。

还可以描述其他一些细节，但这些可能是最重要的了，而且这些已经足以说明封装的概念了。

封装的原则表明，在向一个想学习驾驶的人描述一辆汽车时，应该提供上面列出来的一些内容。在编程环境中，封装也有着相同的含义。这就意味着，编写一段软件时使用的描述方式，应该是要告知其他程序员如何使用这个软件，但要省略软件工作的所有细节。尤其是，如果软件片段有10页长，那么交给另一个使用此软件的程序员的软件描述部分应该要比10页短得多，可能只有半页。当然，只有以一种适宜于这种较短描述的方式来编写软件时，才可能用这么短的篇幅来描述这个软件。

注意，封装隐藏了“小盒子”内部的具体细节。因此，通常又将封装称为**信息隐藏**（information hiding）。

另一种可能会有帮助的类比是，一辆汽车中有些东西是可见的，比如踏板和方向盘，而另外一些东西是隐藏在罩子下面的。将汽车封装起来，就可以将这些细节隐藏在罩子下面，只有驾驶汽车所需的控制部分是可见的。同样，软件片段也应该被封装起来，这样就可以将细节隐藏起来，只有必要的控制部分是可见的。

封装简化了那些用封装好的软件去编写其他软件的程序员的工作，因此，封装的概念很重要。

快速参考：封装

封装是一种过程，这个过程隐藏了软件是如何编写出来的所有细节，仅告诉程序员使用这个软件时需要了解的内容。换句话说，封装是一种描述类或对象的过程，描述时只给出了程序员使用类或对象时必需的信息。

1.2.3 多态性

多态性（polymorphism）来自一个表示“很多形态”的希腊语单词。多态性的基本思想就是允许相同的程序指令在不同的上下文中具有不同的含义。自然语言中经常会出现多态性，多态性在编程语言中的应用使编程语言更像人类的语言。例如，自然语言指令“去参加你最喜欢的运动”对不同的人就有着不同的含义。对某个人来说，它意味着去打篮球；对另一个人来说，它意味着去踢足球。

在Java这样的编程语言中，多态性意味着：根据执行动作的对象种类的不同，一个作为指令使用的方法名可以引发不同的动作。例如，可能有一个名为output的方法，负责输出对象中的数据。但它要输出哪些数据以及输出多少数据项，则取决于执行动作的对象类型。还会进一步解释多态性，本节的简要介绍会给出一些总体概念。（第7章将对多态性进行更完整的介绍。）

如果在自然语言中经常出现多态性，那么，为什么在编程语言中它就变得很重要了呢？这是因为早期的编程语言中很少具有多态性；将多态性引入编程语言后，程序更易读、更易懂，因此，多态性很重要。

快速参考：多态性

在Java这样的编程语言中，**多态性**意味着：根据执行动作的对象种类的不同，一个作为指令使用的方法名可以引发不同的动作。

1.2.4 继承

继承（inheritance）指的是类的组织方式。这个名字来自于特征继承的概念，比如眼睛颜色和头发颜色等特征的继承，但是，从分类系统的角度来考虑，概念可能会更清晰一些。图1-4显示了这样的一个系统实例。注意，每一层的分类都更加具体：Vehicle类包含了Automobile、Motorcycle和Bus类，Automobile类包含了Family Car和Sports Car类。

Vehicle类具有某些属性，比如有轮子。Automobile、Motorcycle和Bus类“继承”了有轮子这个属性，但添加了更多的属性或限制。例如，一辆Automobile有4个轮子，一辆Motorcycle有2个轮子，而一辆Bus至少有4个轮子。

注意，越在图的上面，类的包容性就越大。一辆School Bus是一辆Bus。因为它是一辆Bus，所以也是一辆Vehicle。但是，一辆Vehicle并不一定是一辆School Bus。一辆Sports

Car是一辆Automobile，也是一辆Vehicle，但一辆Vehicle不一定是一辆Sports Car。

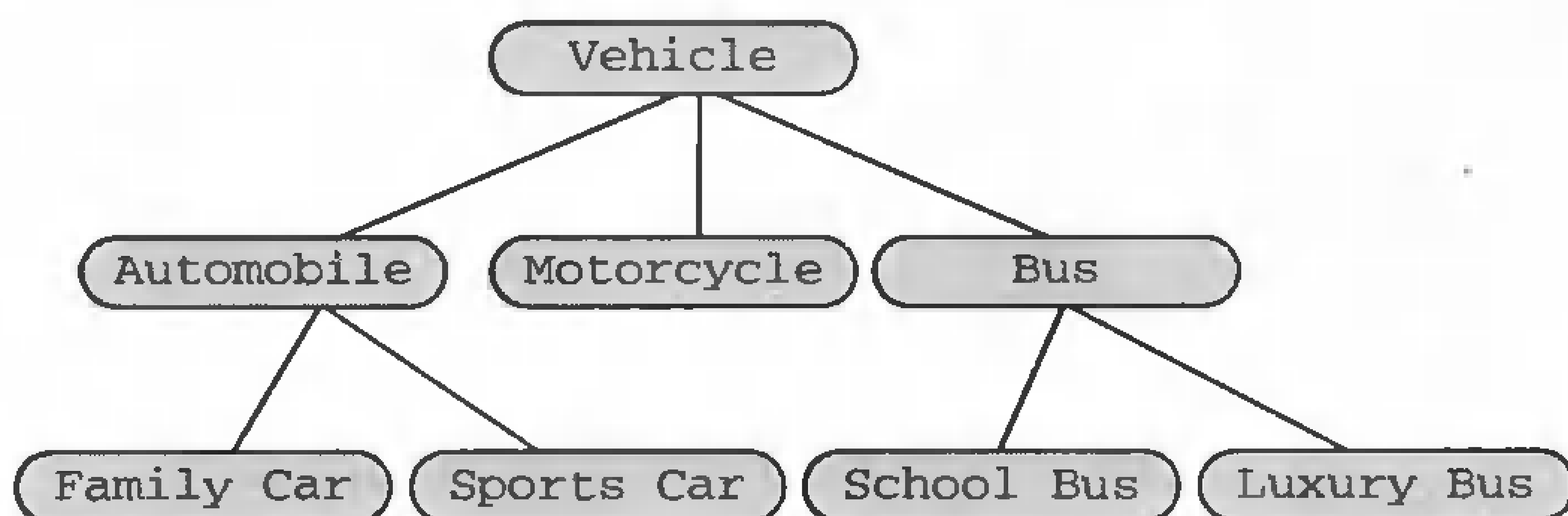


图1-4 继承层次结构

在Java这样的编程语言中，就是按照刚才所描述的方式用继承来组织类的。这种组织方式的优点是，程序员不用为每个类重复相同的编程指令集。例如，所有对每辆Vehicle都为真的属性，比如“有一台发动机”，都只需要描述一遍，并由Automobile、Motorcycle和Bus类继承。如果没有继承，每个Automobile类、Motorcycle类、Bus类、School Bus类、Luxury Bus类等都要将“有一台发动机”这样的描述重复一遍。

对面向对象编程和Java语言来说，继承都是非常重要的。但是，在没有具体编程示例的情况下，理解这个概念有些困难。第7章将更完整、更清晰地解释继承的概念。

快速参考：继承

在Java这样的编程语言中，**继承**是一种类的组织方式，通过这种方式，属性只需定义一次，就可适用于整个类的集合。

1.2.5 如果了解其他一些编程语言

如果Java是你学习的第一种编程语言，可以跳过本小节。如果你了解其他一些编程语言，本小节的讨论有助于通过所了解的事物来理解对象的概念。如果你很熟悉其他面向对象编程语言，比如C++、Smalltalk、Borland的Turbo Pascal或Delphi，那么对什么是对象、方法和类就会有很清晰的概念。尽管其他语言会用*函数*（function）或*过程*（procedure）来表示和*方法*（method）一样的内容，但在所有的面向对象编程语言中，这些概念基本上都是相同的。你熟悉的也可能是一种没有使用对象和类的比较老的编程语言，在这些语言中，可以用其他比较老的编程结构来描述对象。如果你了解变量和函数，就可以把对象当成一个具有多份数据和自己的函数的变量。方法与较老的编程语言中的*过程*或*函数*实际上是一回事。

1.2.6 算法

对象通过执行动作（即方法）进行交互。作为程序员，要通过给出执行动作的指令来设计这些动作。设计动作的过程中最难的不是弄清楚如何用Java（或使用的任何一种编程语言）来表示解决方法，而是提出一个执行动作的计划或策略。这种策略通常是以算法的形式来表示的。

算法（algorithm）是一个用来解决问题的指令集。要想成为一个合格的算法，这些指令的表述必须非常完整和精确，使人只需遵循这些指令，而无需填充任何细节或者做任何指令

中没有完全指定的决定。算法可以用自然语言或Java这样的编程语言来表述。但是，使用算法一词时，通常意味着指令是用自然语言表示的。

举一个例子可能能够帮助澄清算法的概念。第一个示例算法是用来确定一个项目列表的总费用。比如，这个项目列表可能是一个包含了每个项目价格的购物单。然后，就可以用算法来确定列表中所有项目的总费用了。

确定一个项目列表的总费用的算法

1. 在黑板上写上数字0。
2. 对列表中的每个项目完成下列操作：
   - 将项目的费用与黑板上的数字相加。
   - 用这次加法的结果取代黑板上原来的数字。
3. 宣布答案就是写在黑板上的那个数字。

这个算法用黑板来存储中间结果。大多数算法都需要存储一些中间结果。如果算法是用Java语言编写的，并运行在一台计算机上，中间结果就存储在计算机的主存储器中。

快速参考：算法

算法是一个用来解决问题的指令集。要想成为一个合格的算法，这些指令的表述必须非常完整和精确，使人只需遵循这些指令，而无需填充任何细节或者做任何指令中没有完全指定的决定。

自测题

11. 什么是方法？
12. 类和对象之间有什么关系？
13. 同一个类的所有对象都具有相同的方法吗？
14. 什么是封装？
15. 什么是信息隐藏？
16. 什么是多态性？
17. 什么是算法？

1.2.7　可复用组件

第一次开始编写程序时，很容易有这样一种印象：生成的每个程序都是需要从头开始设计的、完全独立的项目。但是，这并不是生成优秀软件的方式。大多数程序都是通过将已存在的组件结合起来而创建的。这样可以节省时间和金钱。而且，复用的软件可能已经被用过很多次了，很可能经过了很好的测试，因此会比新创建的软件更加可靠。

例如，一个高速公路仿真程序中可能会包含一个新的高速公路对象，以仿真新的高速公路设计方案，但它很可能会使用为其他程序设计的汽车类来仿真汽车。为了确保程序中使用的类都可以方便地复用，必须将它们设计为可复用的。你必须确切地指定那个类的对象如何与其他对象交互。这是前面讨论过的封装的原则。但封装并不是需要遵循的唯一原则。设计类时还要使对象是通用的，而不能为特定的程序以专门的方式来设计类。例如，如果你的程序中要求所有的仿真汽车都只能向前开，但是因为其他一些仿真程序可能需要汽车能够后退，

所以还是应该在汽车类中包含反方向的移动。(我们将在学习了一些Java语言的细节并有一些实例可用之后，再回过头来讨论可复用性的话题)。

1.2.8 测试与调试

编写一个正确程序的最好方法就是仔细地设计程序所需的对象以及对象中会用到的那些方法的算法，然后仔细地将所有这些内容转换成Java代码（或所使用的任何一种编程语言）。换句话说，消除错误的最好方法是一开始就避免出现错误。但是，无论多么仔细，程序中仍然可能包含一些错误。编写完一个程序之后，应该对其进行测试，看看它是否能够正确地执行，然后对找到的所有错误进行修正。

程序中的错误称为bug。因此，消除程序中错误的过程被称为**调试**（debugging）。bug（或错误）的类型一般有3种：语法错误、运行时错误和逻辑错误。下面按这个顺序进行讨论。

语法错误（syntax error）是程序中语法方面的错误。编写程序有着非常严格的语法规则，如果违反了其中的一条规则——比如，省略了一个必需的标点符号——就会产生一个语法错误。编译器会捕获语法错误，输出一条错误消息，通知你它发现了错误，并指出它认为是个什么错误。如果编译器说有语法错误，很可能确实有错误。但是，编译器只是在猜测那是个什么样的错误，因此，编译器对问题的诊断可能是不正确的。

快速参考：语法

正确地编写一个程序或部分程序所需遵循的规则（一种编程语言的语法规则）被称为这种语言的**语法**。

在程序运行时检测出来的错误称为**运行时错误**（run-time error）。如果程序中有运行时错误，计算机会在程序运行时输出一条错误消息。错误消息可能很好理解，也可能不太好理解，但至少它让你知道出现错误了。有时，它甚至能确切地指出问题出在什么地方。

如果程序的基础算法出现了错误，或者用Java编写的程序不正确但仍然合法，那么，程序在编译和运行时都不会抛出任何错误信息。你编写了一个合法的Java程序，但并没有编写出你想要的程序。程序能够运行并产生输出，但输出不正确。在这种情况下，程序中就含有**逻辑错误**（logic error）。例如，错误地将减号写为加号，这就是个逻辑错误。这样的程序可以编译并运行，而且不产生错误信息，但程序会给出错误的输出。逻辑错误是最难定位的错误种类，因为计算机不会为你提供任何错误信息。

▲ 易犯错误：对付“易犯错误”

任何编程语言都有一些细节会以令人惊异或难以对付的方式让你犯错误。这类问题通常被称为缺陷，但一个更生动也更常用的词是**易犯错误**（gotcha）。这个词源于这样的事实：那些问题、缺陷，或易犯错误就像等着捉住你的陷阱一样。当你掉进陷阱时，陷阱就“抓住你了（got you，或者更常见地读成“gotcha”)”。

本书中有些像这里一样标以“易犯错误”的段落，会提醒你注意很多最常见的易犯错误，并指出应该如何避开或对付它们。 △

▲ 易犯错误：隐藏的错误

程序在编译及运行时没有出错，甚至给出了看起来合理的输出，并不意味着程序是正确的。应该用一些知道会输出什么结果的测试数据来运行程序。要做到这一点，就要选择一些可以通过铅笔和纸、通过查找答案或者通过一些其他方法计算出正确答案的数据。即使这样的测试也无法确保程序是正确的，但所做的测试越多，对自己的程序就会越有信心。△

自测题

18. 什么是语法错误？
19. 什么是逻辑错误？
20. 什么样的错误可能会产生错误信息，警告你程序中有错？
21. 假设你编写了一个程序，要用这个程序来计算一个指定日期（如2004年10月1日）是星期几（星期日、星期一等）。现在假设你忘记考虑闰年的问题了，于是，程序中会存在一个错误。请问这是哪种类型的程序错误？

1.3 Java简述

Java（名词）是印度尼西亚的一个岛，面积48 842平方英里，位于印度洋和爪哇海之间。
java（名词）也指煮制的咖啡（非正式用语）。

——《美国传统英语字典》（第1版）

本节介绍Java语言的一些特性，并研究一个简单的Java程序。

1.3.1 Java语言的历史

人们普遍认为Java是一种用于因特网应用程序的编程语言。但是，本书及很多其他的书和人，都将Java作为一种通用的编程语言，可以用于和因特网没有任何关系的地方。Java刚出现的时候并不是针对其中任何一种情形的，但它最终发展成为一种对这两种情形都适用的语言。

Java的历史可以追溯到1991年，Sun公司的James Gosling和他的团队开始设计一种会成为Java（尽管那时候还不这样称呼它）的新的编程语言的第一个版本。Java第一个版本的目标是成为一种为烤箱和电视这样的家用电器编程的编程语言。这听起来像是一项简单的工程任务，但实际上却是一项非常有挑战性的工作。家用电器是由大量形式多样的计算机处理器（芯片）控制的。Gosling和他的团队设计的语言要能工作在所有这些不同的处理器上。而且，家用电器通常都是不太贵的东西，所以制造商可能不愿意投入大量的时间和金钱开发复杂的编译器（编译器是将设备语言程序翻译成处理器可以理解的语言的程序）。在这个团队设计的设备语言中，以及从这个设备语言演变而来的Java语言中，程序首先会被翻译成一种**中间语言**（intermediate language），这种中间语言对所有的设备（或所有的计算机）都是一样的。然后，由一个小型的、容易编写的因此也就不太贵的程序将中间语言翻译成特定设备或计算机使用的机器语言。如1.1.4节所述，中间语言被称为Java字节码，或简称为字节码。用Java的第一

个版本对设备进行编程的计划没有受到设备制造商们的欢迎，但是，故事并没有结束。

1994年，Gosling意识到，用他的语言来开发一个可以在因特网上运行（Java）程序的Web浏览器是非常理想的。Sun公司的Patrick Naughton和Jonathan Payne开发了一个Web浏览器，并演变成今天大家所熟知的HotJava。这就是Java与因特网联系的开始。1995年秋，Netscape公司决定在其最新的Web浏览器版本中支持Java程序的运行。其他与因特网有关的公司也紧跟其后，并开发了一些适应Java程序的软件。

常见问题：为什么这种语言被称为“Java”？

关于如何起了Java这个名字的问题，并没有什么有趣的答案。现在的习惯是，为编程语言取名的过程与父母为他们的孩子取名的过程基本上是一样的。编程语言的创建者只是挑选一个对他们来说听起来还不错的名字。这个语言最初的名字是“Oak”。后来，语言的创建者们意识到已经有一种名为Oak的编程语言了，这样，他们就需要另取一个名字，于是选择了“Java”。关于“Java”这个名字的起源人们会听到一些相互矛盾的解释。一种传统的、可能也比较可信的说法是，在开了一个试图为这种语言取个新名字的会议，但毫无结果之后，研发小组出去喝咖啡，而接下来发生的事情，都已载入历史。

1.3.2 applet

有两种类型的Java程序：applet和应用程序。**应用程序**（application）就是一个常规的程序。applet听起来像是个小苹果似的，但这个名字表示的是小应用程序的概念。applet和应用程序基本上是一样的，区别就在于应用程序像任何其他程序一样是在你的计算机上运行，而applet则要发送到因特网上的另一个地方，在那里运行。

一旦学会了如何设计及编写这两种程序中的一种，不管是applet还是应用程序，学习另一种程序的编写就很容易了。本书的组织方式使你可以按照自己的意愿决定在applet上花多少功夫。大多数章节的重点都放在应用程序上，但是从本章开始，每章末尾处选读的“图形编程补充”小节都会给出与applet相关的资料。本书姊妹篇《Java程序设计与问题解决：高级篇(第4版)》还对applet进行了详细的讨论。你可以选择学习“图形编程补充”小节，较早地开始了解applet，也可以阅读本书姊妹篇《Java程序设计与问题解决：高级篇（第4版)》来学习applet。如果不想马上开始学习applet，但又想看一些简单的applet示例，可以只阅读1.4节。

1.3.3 第一个Java应用程序

图1-5显示了本书的第一个Java程序。在程序下面，显示了有人运行这个程序并与之进行交互时，可能产生的屏幕对话。与程序进行交互的人被称为**用户**（user)。用户输入的文本用彩色文字表示。如果你运行该程序（也应该运行一下这个程序)，在计算机显示器上，程序显示的文本和你输入的文本将是同一种颜色的。用户可以是编写程序的人，也可以不是。在编程课上，他们很可能是同一个人，但在现实应用中，他们通常都是不同的人。

编写程序的人称为**程序员**（programmer)。本书的目的就是培养你成为一名程序员，而你首先需要了解的事情之一就是不能期望程序的用户知道你希望他们做什么。因此，你的程序必须像屏幕对话示例中所做的那样，为用户提供一些可理解的指令。

```
import java.util.*;          ← 包含Scanner类的包（库）的名字。

public class FirstProgram    ← 程序名。
{
    public static void main(String[] args)
    {                                                  将输出送到屏幕上。
        System.out.println("Hello out there.");   ←
        System.out.println("I will add two numbers for you.");
        System.out.println("Enter two whole numbers on a line:");

        int n1, n2;   ← 声明n1和n2是装有整数（whole number）的变量。
                         创建对象，使程序可以接收键盘输入。
        Scanner keyboard = new Scanner(System.in);
        n1 = keyboard.nextInt();
        n2 = keyboard.nextInt();     ← 从键盘读入一个整数。

        System.out.println("The sum of those two numbers is");
        System.out.println(n1 + n2);
    }
}
```

屏幕对话示例

```
Hello out there.
I will add two numbers for you.
Enter two whole numbers on a line:
12 30
The sum of those two numbers is
42
```

图1-5　Java程序示例

现在，我们只是希望通过对图1-5显示的示例程序的简要和非正式的描述，使你对Java语言有些概念。第2章和第3章将对程序中使用的Java特性的细节进行介绍。*如果在首次阅读中对程序的某些细节不完全清楚，也不用担心。*这些细节会在第2章和第3章中变得清晰起来。这里只是简介。

下面显示的是程序中的第一行，这一行告诉编译器这个程序使用了java.util（Java utility（实用程序）的缩写）包。

```
import java.util.*;
```

包（package）是已经定义好的类库。（类是可以在程序中使用的软件片段。）正如你很快将会看到的那样，这个程序使用了Scanner类，而Scanner类是在java.util包中定义的。

我们暂时忽略下面几行出现在程序起始处的代码：

```
public class FirstProgram
{
    public static void main(String[] args)
    {
```

这些起始行为程序建立了一个上下文环境，但现在还不用操心这些代码。你可以认为这

是Java书写“开始一个名为FirstProgram的程序”的方式。

下面3行是程序采取的第一批动作：

```
System.out.println("Hello out there.");
System.out.println("I will add two numbers for you.");
System.out.println("Enter two whole numbers on a line:");
```

这几行中每行都以System.out.println开头。每行都将圆括号中给出的引用字符串输出到屏幕上去。例如：

```
System.out.println("Hello out there.");
```

这行代码会将下面的一行内容输出到屏幕：

```
Hello out there.
```

你可以认为System.out.println是以一种有趣的方式在说“输出圆括号中显示的内容。”但是，我们可以告诉你一点儿这里发生的事情。

就像前面讨论的那样，Java程序是通过让对象执行动作来工作的。对象执行的动作称为方法。System.out是一个负责将输出发送到屏幕的对象；println是这个对象为了将圆括号中的内容发送到屏幕而执行的一个方法。当对象用方法来执行一个动作时，就称它**调用**（invoke或call）了这个方法。在Java程序中，可以写下对象名，接着写一个句点（在计算机术语中称为一个**点**），然后写上方法名和一些可能包含（也可能不包含）内容的圆括号，这样就写下了一个方法调用。圆括号中的项目被称为**实参**（argument），为方法提供了执行其动作时所需的信息。在前面这3行代码中，使用的方法为println。方法println会向屏幕写一些内容，由实参（引号中的一个字符串）来告诉该方法应该写些什么。

快速参考：类、对象和方法

Java程序是通过让某种调用**对象**的事物执行动作来工作的。这种动作称为**方法**。同一种类型的所有对象被称为属于同一个类。因此，**类**就是对象的种类。（如果想了解更多有关类和对象的细节，见1.2.1节。）当对象执行某个指定方法的动作时，就称它**调用**了这个方法。

下面是程序中的下一行代码，说明了n1和n2是变量名：

```
int n1, n2;
```

变量（variable）可以存储一份数据。int说明数据必须是整数。

接下来的一行代码创建了一个对象，以便从键盘输入数据：

```
Scanner keyboard = new Scanner(System.in);
```

这行代码创建了一个Scanner类的对象，名为keyboard。程序可以用这个名为keyboard[①]的对象读入键盘输入的数据。第2章将对这行代码进行详细的解释。

接下来的一行代码读入键盘上输入的一个数字，并将这个数字存储在变量n1中：

```
n1 = keyboard.nextInt();
```

再下一行代码和这一行基本相同，只是它读入的是键盘上输入的下一个数字，并将第2个数字存储在变量n2中。因此，如果用户像示例对话所示的那样，输入了数字12和30，那么，这两行代码就会将数字12存储在变量n1中，将数字30存储在变量n2中。

下面是接下来的两行代码，输出一个解释性的短句以及存储在变量n1和n2中的数字之和：

① 正如稍后可以看到的那样，可以用其他名字来代替keyboard，但这不是现在需要关心的。

```
System.out.println("The sum of those two numbers is");
System.out.println(n1+n2);
```

注意，第二行代码中包含的是表达式n1+n2，而不是引号中的字符串。表达式n1+n2计算出了存储在变量n1和n2中的数字之和。当这样的输出语句中包含了一个数或一个数值表达式（而不是引用字符串）时，就会将这个数输出到屏幕。因此，在图1-5所示的对话中，这两行代码产生的输出如下所示：

```
The sum of those two numbers is
42
```

在这个程序中剩下的唯一需要解释的就是每行末尾的分号和程序末尾的花括号了。分号就像英文句子中的句号一样，充当结束的标点。分号通知计算机一条指令的结束。指令被称为**语句**（statement）。末尾的花括号仅表示程序的结束。

当然，Java程序的每一部分应该如何编写都有严格的规则，这些规则形成了Java语言的语法，就像英语的语法规则一样，但Java的语法规则要更严格一些。编程语言（或者任何语言）的语法规则被称为这种语言的**语法**（syntax）。

快速参考：方法调用

方法是对象可以执行的动作。请求一个对象执行一个方法的动作，就称为调用那个方法。在Java程序中调用一个方法时，要写下对象名，接着写一个点，然后写上方法名，后面再跟上用圆括号括起来的实参。**实参**是传递给方法的信息。

举例：

```
System.out.println("Hello out there.");
n1 = keyboard.nextInt();
```

在第一个例子中，System.out是对象，println是方法，"Hello out there"是实参。如果有多个实参，实参之间以逗号分隔。

在第二个例子中，keyboard是对象，nextInt是方法。方法nextInt没有实参，但圆括号还是需要的。

在一个程序中，方法调用后面通常都要跟一个分号。

常见问题：为什么输入需要import，而输出却不需要呢？

考虑图1-5中的下列代码行：

```
import java.util.*;
```

有了这行代码，我们才能像下面这样接收键盘输入：

```
n1 = keyboard.nextInt();
```

为什么不需要某种类似import的语句，以实现

```
System.out.println("Hello out there.");
```

这样的屏幕输出呢？答案是相当乏味的。Java中自动导入了包含了屏幕输出定义及屏幕输出代码的包。

自测题

22. 在Java程序中使用下列语句时，会向屏幕输出什么内容？

```
System.out.println("Java is great!");
```

23. 给出一条或多条可以用在Java程序中向屏幕输出下列内容的语句：
```
Java for one.
Java for all.
```
24. 假设mary是一个名为Person的类的对象，假设increaseAge是Person类的方法，它有一个整型实参。怎样用实参5来调用对象mary的方法increaseAge呢？方法increaseAge会对mary中的数据进行修改，以便仿真mary长大了5岁。
25. 图1-5所示的程序中出现的下列代码行是什么意思？
```
n1 = keyboard.nextInt();
```
26. 编写一个完整的Java程序，用System.out.println在程序运行时将下列信息输出到屏幕上：
```
Hello World!
```
程序没有做任何其他事情。注意，要编写程序，并不需要完全理解程序的所有细节。你可以简单地按照图1-5中的程序模式来编写。（最终还是希望你能够理解所有的细节，但那还需要再经过几章的介绍。）

1.3.4 编译一个Java程序或类

一个Java程序可以被分成一些较小的部分，称为类，而且通常每个类定义都在一个单独的文件中。在运行一个Java程序之前，必须将这些类翻译成计算机能够理解的语言。这种翻译过程被称为编译。（类的有关内容参见1.2.1节。编译的有关内容参见1.1.3节。）

一个Java程序可以包含任意数量的类定义。图1-5中的程序只包含一个类。该类被命名为FirstProgram。Java中的每个程序都是一个类，也是一个程序。

实际上，图1-5所示的程序中还使用了另外两个类：System和Scanner。但是，这两个类都是由Java提供的，你不需要为编译它们而操心。一般来说，那些作为Java的一部分提供的类是不需要编译的。通常你只需要对自己编写的类进行编译。

在编译一个Java程序之前，程序中使用的每个（由程序员编写的）类定义都应该放在一个单独的文件中。而且，除了在文件名末尾处添加.java之外，文件名应该与类名相同。图1-5中的程序是一个名为FirstProgram的类，因此应该将其放在一个名为FirstProgram.java的文件中。

在运行图1-5所示的程序之前，必须对文件FirstProgram.java中的类FirstProgram进行编译。

编译并运行Java程序的最简单的方式就是使用IDE（集成开发环境，Integrated Development Environment）。IDE是一种特殊的环境，它将文本编辑器和一些用来编译、运行Java程序的命令结合在了一起。如果你在使用IDE，其中很可能会有一条可以用来编译Java类或Java程序的菜单命令（可以用相同的命令去编译任何类型的Java文件）。Sun公司的NetBeans和TextPad环境就是这样的两个IDE，Sun公司的NetBeans可以在提供Java编译器的网站上找到，而TextPad环境则更容易使用。需要查看一下你的文档以确定这条命令到底是什么，但肯定是很简单的。（在TextPad环境中，就是Tools菜单中的Compile Java命令。）可能需要根据下面一段描述来配置编译命令。

如果要输入一条单行的命令来编译Java程序，也很容易。下面将对一些用于Sun公司发布

的Java系统的命令进行描述。

编译一个Java类的命令为javac -source 1.5，后面跟着包含类的文件名。假设你想编译一个名为MyClass的类。应该将这个类存放在一个名为MyClass.java的文件中。要想编译这个类，只要向操作系统发送下列命令即可[1]：

```
javac -source 1.5 MyClass.java
```

因此，可以通过下列命令来编译图1-5中的程序：

```
javac -source 1.5 FirstProgram.java
```

记住，如果有一个可以用菜单命令来进行编译的环境，你就会发现使用菜单命令要比使用前面的命令容易得多。

关于-source 1.5有一些额外要注意的。首先，不是所有的程序都需要包含-source 1.5，以后的Java版本可能也不需要包含它，但即使不需要，包含它也不会有什么问题。第二，如果使用的是可以用菜单命令进行编译的环境，可能需要对编译命令进行配置，使其包含-source 1.5[2]。设置-source 1.5的选项说明，要以一种能够使用版本5.0中所有新特性的方式进行编译。在Java编译器今后的版本中，这些细节可能会有所改变。如果在编译程序时遇到了问题，可以到Sun公司的Web站点http://java.sun.com/或者作者的Web站点上去查询。

编译一个Java程序（或Java类）时，编译器生成的经过翻译的程序（或其他类的）版本被称为字节码。得到的程序（或其他类的）字节码被放在一个同名的文件中，只是后缀从.java变成了.class。所以，在对文件MyClass.java中名为MyClass的类进行编译时，得到的字节码会被存放在一个名为MyClass.class的文件中。编译一个名为FirstProgram.java的类文件时，得到的字节码会被存放在一个名为FirstProgram.class的文件中。（关于字节码的内容参见1.1.4节。）

1.3.5 运行一个Java程序

我们的第一个Java程序仅编写了一个类。然而，一个Java程序可以包含任意数量的类，但当你运行一个Java程序时，只是运行了你认为是程序的那个类。你可以识别出这个类，因为这个类中会包含与

```
public static void main(String[] args)
```

完全相同或非常类似的一些单词。

这些单词很可能（但不是必须）出现在文件起始处的某个地方。需要查找的关键词是public static void main。在某些情况下，这一行其余部分的拼写可能会稍有不同。

如果你在使用IDE，就会有一条可以用来运行Java程序的菜单命令。可以查看一下你的本地文档，确认这条命令到底是什么。（在TextPad环境中，是File菜单中的Run Java

---

① Sun公司修改了他们的版本编号方式。版本5.0原来被称为版本1.5。这就是-source 1.5中使用1.5的原因。将来这些细节有可能会发生变化。如果在编译程序时遇到了问题，可以到sun公司的网站http://java.sun.com/以及/或者作者的Web站点上去搜索。

② 对TextPad环境来说，需要按照下列方式添加-source 1.5。选择Configure菜单上的Preferences命令。如果Tools左边有个加号，点击+号，它就会变成一个减号，同时会出现更多的文本。如果Tools旁边已经是减号了，就不要点击这个减号了。接着，点击-Tools下的Compile Java。标记为Parameters的窗口中应该包含-source 1.5 $File；如果没有，就将它改成-source 1.5 $File。最后，点击Apply按钮，然后点击OK按钮。

Application命令。）

如果要输入一条单行的命令，（在大多数系统中）都可以通过在java命令后面跟上你认为是程序的那个类的名字，来运行一个Java程序。例如，要运行图1-5中的程序，应该使用下面的单行命令：

```
java FirstProgram
```

注意，在运行程序时，使用的是FirstProgram这样的类名，不需要任何.java或.class后缀。而且，要记住，如果有一条可以运行Java程序的菜单命令，那会是运行Java程序的更简单的方法。

运行Java程序时，实际上是在对经过编译的程序运行Java字节码解释器。

在前面的讨论中，假设你已经安装了Java编译器和其他系统软件；否则，就需要安装Java编译器和相关软件，可以查阅与软件配套的手册。

自测题

27. 假设在一个文件中定义了一个名为SuperClass的类。这个文件应该叫什么名字？
28. 假设对SuperClass类进行了编译。装有得到的字节码的文件应该叫什么名字？

1.4 图形编程补充（选读）

祝你今天过得愉快。

——常见告别用语

本书中的每一章都有一个本节这样的选读小节，其中涵盖了那些包含各种图形显示的程序所需的材料。通常，图形都是在applet中显示的。如第2章所示，除了applet之外，图形通常也是视窗界面的一部分。这些小节是选读的，因为有些人希望在程序员（比如你自己）掌握了更多的基本资料之后再学习图形的相关内容。为了较早地开始学习图形编程，我们得借助于一些“魔法公式”——就是一些将告诉你如何使用但要在本书稍后才会进行完整解释的代码。

这些图形编程补充材料是选读的；可以跳过，也可以阅读。但是，这些“图形编程补充”材料是互相依存的。如果想学习某一章的“图形编程补充”小节，就要先学习前面各章中所有或大部分的“图形编程补充”小节。

因为与applet有关的内容和这里提供的图形编程资料都用到了类和方法，本节我们就开始介绍一些与类和方法有关的背景资料。

1.4.1 对象和方法

对象是一些存储了数据并能执行动作的实体。在第2章中，我们将会用对象来存储数据，又会用它来执行动作。本章只用对象来执行动作。对象执行的动作称为**方法**。前面已经使用过对象和方法了。比如，System.out就是一个对象，println就是对象System.out可以执行的一个方法（动作）。让一个对象执行一个方法的方式是写下对象名，后面跟一个点和方法名。比如：

```
System.out.println("Hello");
```

因为这个方法执行是一条Java语句，所以，我们用一个分号来结束这行方法执行代码（也称为一次方法调用）。

当代码中包含了类似上述的方法调用时，点前面的对象，在这个例子中就是System.out，被称为**调用对象**（call an object），例如：

```
System.out.println("Hello");
```

这样，用来执行方法的整个表达式被称为**方法调用**（method call或method invocation），并称调用对象调用了方法。

类是对象的类别。一个类中所有的对象都具有相同的方法，但每次方法调用都可以在圆括号中使用不同的实参。

本节只使用一种类型的对象，这些对象通常称为canvas。canvas对象有很多方法，可以在applet画面中画出很多形状，比如椭圆形。

1.4.2 图形applet示例

图1-6中包含了一个画一张笑脸的applet。下面逐行地对代码进行解释。

```
import javax.swing.*;
import java.awt.*;

public class HappyFace extends JApplet
{
    public void paint(Graphics canvas)
    {
        canvas.drawOval(100, 50, 200, 200);
        canvas.fillOval(155, 100, 10, 20);
        canvas.fillOval(230, 100, 10, 20);
        canvas.drawArc(150, 160, 100, 50, 180, 180);
    }
}
```

得到的GUI

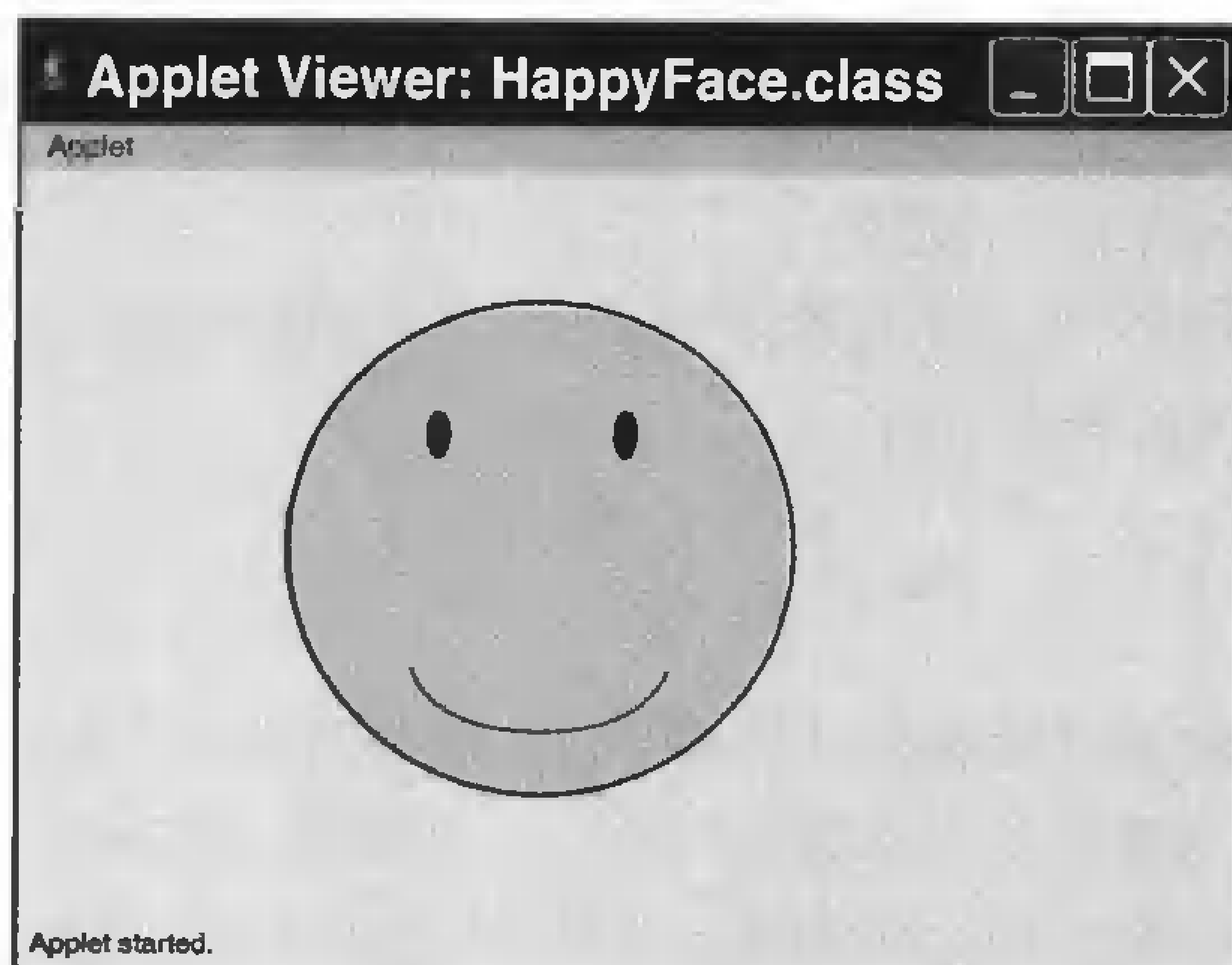


图1-6　画一张笑脸

代码行

```
import javax.swing.*;
```

说明这个程序使用了Swing库（包）。applet使用了Swing库中的软件。

代码行

```
import java.awt.*;
```

说明这个程序还使用了AWT库（包）。applet除了使用Swing库中的软件之外，还经常使用AWT库中的软件。

下一行是

```
public class HappyFace extends JApplet
```

这是applet中类定义的开始。HappyFace是applet的名字，extends JApplet说明我们定义的是一个applet，而不是某种其他类型的类。

以public void paint(Graphics canvas)开头的部分是paint方法定义的开始。paint方法说明了applet中画的是什么图形。applet运行（或者在一个HTML文件中显示applet）时，会自动调用（运行）paint方法。

我们在第4章之前不会讨论方法的定义，但会在这里给出足够的知识，使你能够定义paint方法来进行一些简单的图形编程。public void是必需的，但这里暂不介绍。paint方法执行的动作（都是绘画动作）是一系列包含在花括号中的Java指令（被称为语句）。下面将paint方法定义的这一部分复制了下来：

```
{
    canvas.drawOval(100,50,200,200);
    canvas.fillOval(155,100,10,20);
    canvas.fillOval(230,100,10,20);
    canvas.drawArc(150,160,100,50,180,180);
}
```

花括号中的每行代码都是一条在applet中画一个图形的指令。

快速参考：对象、方法和类

对象是一个程序构造，它有与之相关的数据（即信息），并且可以执行某些动作。对象执行的动作被称为**方法**。**类**是对象的类型或种类。同一个类中的不同对象可以拥有不同的数据。但是，同一个类中的所有对象都具有相同类型的数据和相同的方法。对象执行一个方法的动作时，就称这个对象**调用**了方法。方法调用的语法是：写下对象的名字，后面跟一个点，然后再跟上一对包含了方法调用实参的圆括号。

举例：

```
System.out.println("Hello");
canvas.drawOval(100,50,200,200);
```

在第2个例子中，调用对象是canvas，方法是drawOval，实参是100、50、200和200。

方法调用

```
canvas.drawOval(100,50,200,200);
```

画了一个大圆，这个圆构成了脸部轮廓。前面两个数字说明在哪里画圆。方法drawOval画出的是椭圆。后两个数字给出了椭圆的横轴和竖轴。使用具有相同的横轴和竖轴的drawOval就可以画出一个圆。这些数字使用的单位称为*像素*（pixel）。

下面两个方法调用画出了两个眼睛：

```
canvas.fillOval(155,100,10,20);
canvas.fillOval(230,100,10,20);
```

注意，这次画的是“真的”椭圆，横轴要大于竖轴。还要注意，方法称为fillOval，而不是drawOval，表示画的是一个被填充了的椭圆。

最后一个方法调用

```
canvas.drawArc(150,160,100,50,180,180);
```

画出了嘴。本节中将对实参（圆括号中的数字）的含义进行解释。1.4.3节对所有这些数字进行了解释。

1.4.3 图形的尺寸和位置

Java applet（或者其他的屏幕显示）中所有的尺寸和距离都是以像素（pixel）而不是以英寸或厘米为单位给出的。像素不是英寸或厘米那样的绝对长度单位。在不同的显示器上像素的尺寸可能是不同的，但通常是一个很小的单位。可以认为计算机显示器是由很多小的正方形覆盖的，每个小正方形都可以是任意颜色的。你无法显示任何小于这种小正方形的内容。**像素**就是这样一个正方形；或者，在用来衡量长度时，**像素**就是这些小正方形的边长[①]。因此，像素就是显示器能够显示的最小长度。如果你买过数码相机，那你肯定听说过像素或兆像素（megapixel）的概念。Java applet中使用的像素的含义和描述数码相机照片时使用的像素的含义是一样的，**兆像素**就是100万像素。

图1-7显示的是用来在applet（或者其他类型的Java类视窗画面）内进行图形定位的坐标系。把大的长方形当作屏幕上显示的applet的内部轮廓。坐标系为applet内的每个点分配了两个数字（通常写在圆括号内）。这些点被称为*x*坐标和*y*坐标。因此，用×标识的点的坐标就是（100，50）。100是*x*坐标，50是*y*坐标。*x*坐标就是从方框的左端到那个点的像素数，*y*坐标就是从方框的顶端到那个点的像素数。如果你在某些数学课中学习过*x*坐标和*y*坐标，那么这里的*x*坐标和*y*坐标除了下面一点之外，其余和数学中的坐标是相同的：这个系统中的*y*坐标是正的，并且从点（0，0）开始向下递增。在大多数数学课中，*y*坐标是正的，并且会从点（0，0）向上递增。如果你从来没在数学课中学过坐标系，也不用担心，这里给出的介绍已足够了。

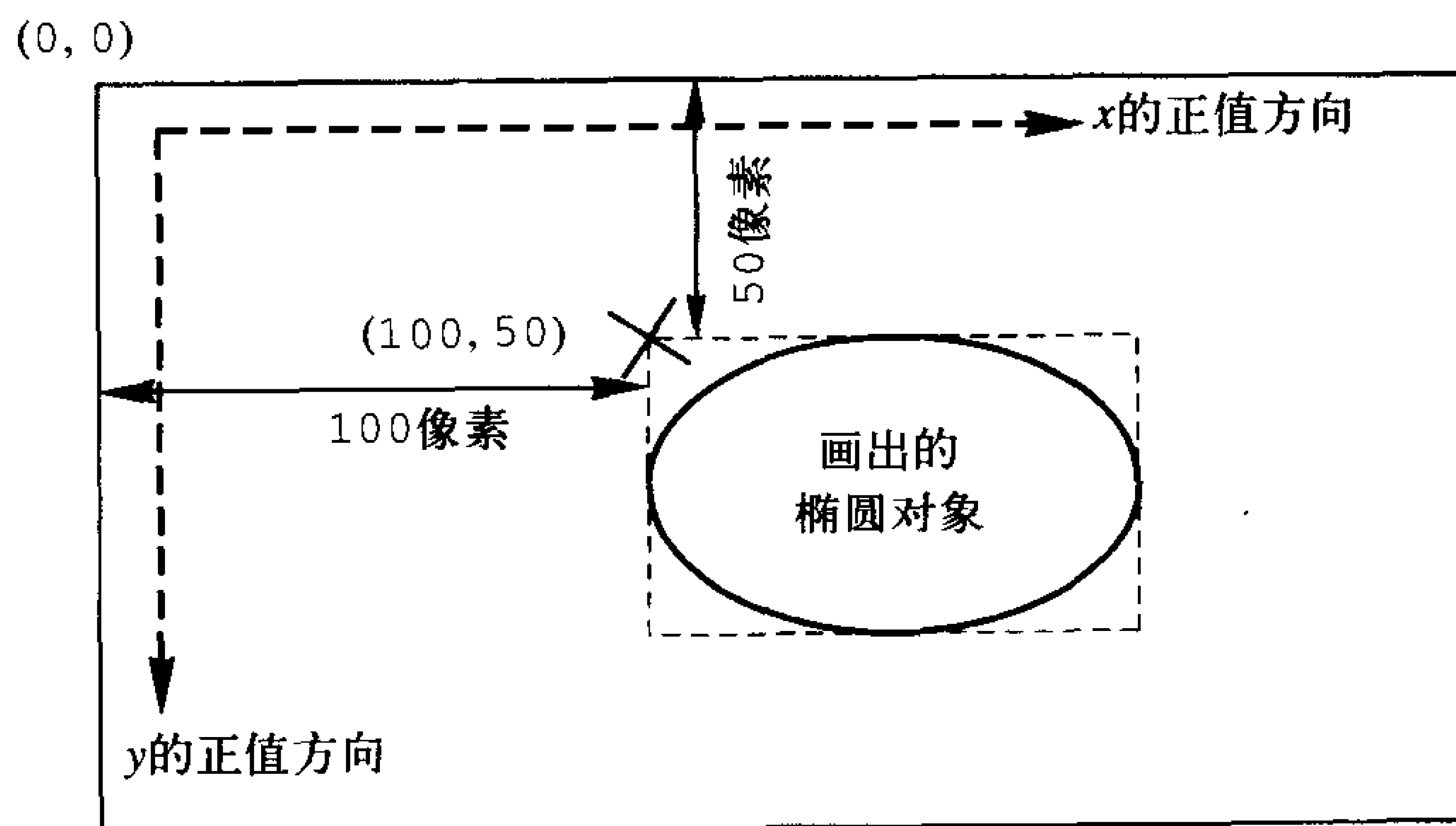


图1-7 显示器坐标系

① 严格地说，像素不一定是正方形，也可以是长方形。但是，在这里不需要讨论这种细节。

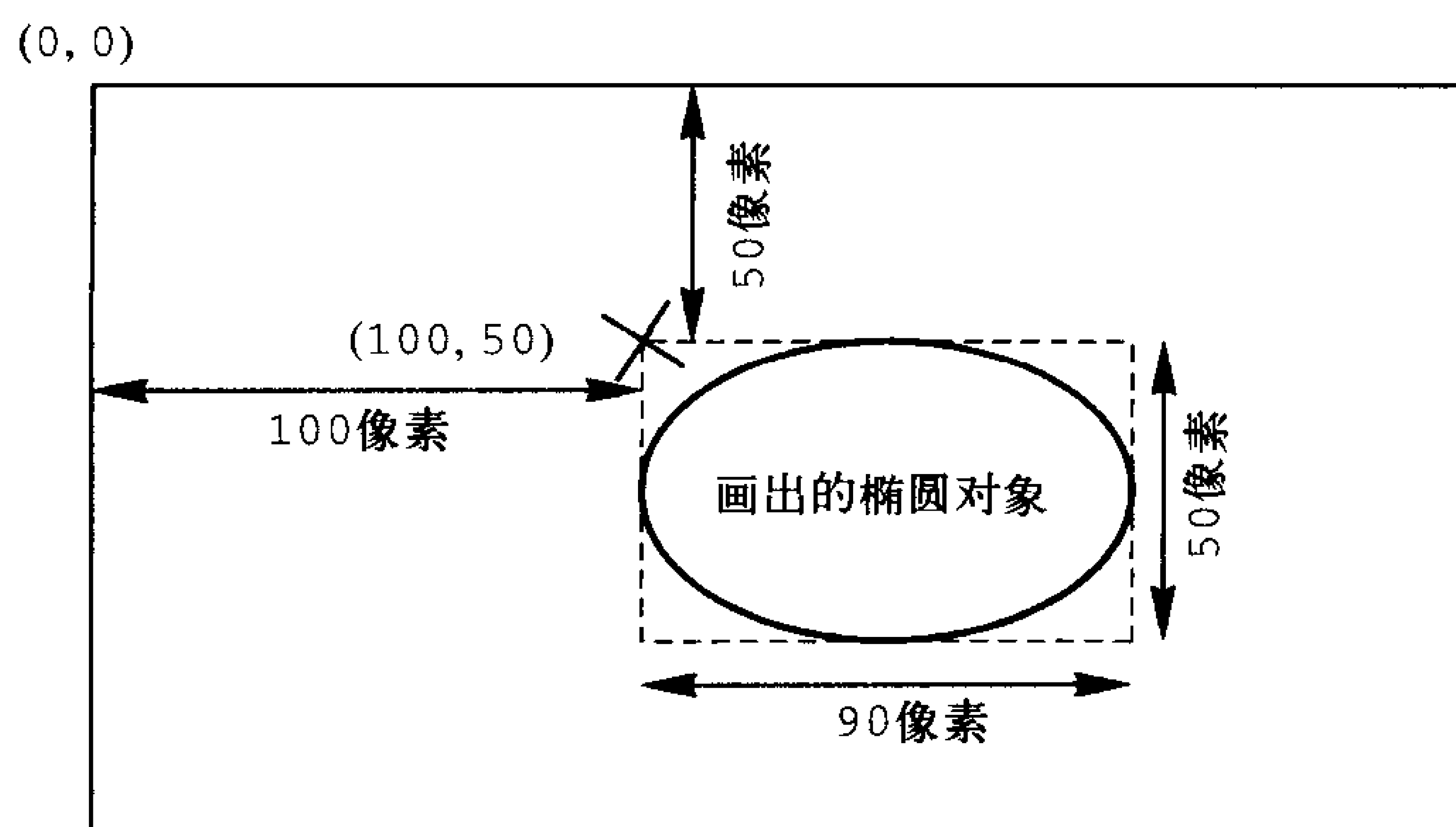
要像图1-7中带×的虚线框标出的那样，在坐标（*x*，*y*）上定位一个长方形，就应该将长方形的左上角放在点（*x*，*y*）上。例如，虚线标出的长方形就位于点（100，50）上。如果要定位的图形不是一个长方形，就用一个尽量小但又能包含这个图形的长方形将其包围起来，以此来对其进行定位。放置这个图形时，使包围它的这个长方形的左上角位于点（*x*，*y*）。比如，在图1-7中，椭圆也被定位在点（100，50）上。如果applet中只包含一个椭圆而不包含长方形，就只有椭圆会显示在屏幕上，但仍然会用一个假想的长方形来对椭圆或其他图形进行定位。

现在，回过头来进一步解释applet代码中的数字。再次将带有数字（作为实参）的代码行复制过来：

```
canvas.drawOval(100,50,200,200);
canvas.fillOval(155,100,10,20);
canvas.fillOval(230,100,10,20);
canvas.drawArc(150,160,100,50,180,180);
```

在所有的情况下，前两个数字给出的都是要画的图形的位置。第一行代码在位置（100，50）上画了一个圆，给出了脸部的轮廓。下面两行在点（155，100）和（230，100）上画了两个椭圆作为眼睛。最后一行在点（150，160）上画了一个笑容。

那么其他数字呢？对前面3行来说，画的都是椭圆，这些数字给出了要画的椭圆的横轴和竖轴。如果横轴和竖轴相等，画出来的就是一个圆。因此，脸部圆形的直径就是200。每个眼睛都是10像素宽，20像素高。图1-8对其进行了说明。最后一行代码中数字的含义将在1.4.4节介绍。



椭圆是用canvas.drawOval(100,50,90,50);画出来的，位于（100,50），横轴为90，竖轴为50

图1-8 画一个椭圆

快速参考：drawOval和fillOval

方法drawOval画出了椭圆的轮廓。

语法：

canvas.drawOval(*x, y, Width, Height*);

画出一个横轴为*Width*像素，竖轴为*Height*像素的椭圆。放置椭圆时，使紧紧包含它的长方形的左上角位于点（*x*, *y*）上。

举例：

```
canvas.drawOval(100, 50, 90, 50);
```

方法fillOval会画出与drawOval相同的椭圆，不过它画的椭圆是填充满了的。

语法：

canvas.fillOval(*x, y, Width, Height*);

举例：

```
canvas.fillOval(100, 50, 90, 50);
```

1.4.4 画弧线

弧线，比如图1-6中笑脸上的笑容，是这样描绘出来的：给出一个椭圆，然后指定用椭圆的哪一部分做弧线。例如，图1-6中的下列语句在笑脸上画出了一个笑容：

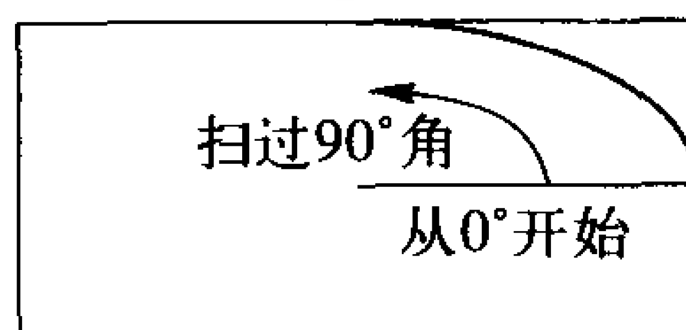
```
canvas.drawArc(150,160,100,50,180,180);
```

实参100和50确定了一个不可见长方形的尺寸。实参150和160确定了这个长方形的位置。长方形的左上角放置在点（150, 160）上。想象一个刚好能放在这个不可见长方形内部的不可见椭圆。最后两个实参指定了这个不可见椭圆中被设为可见的部分。

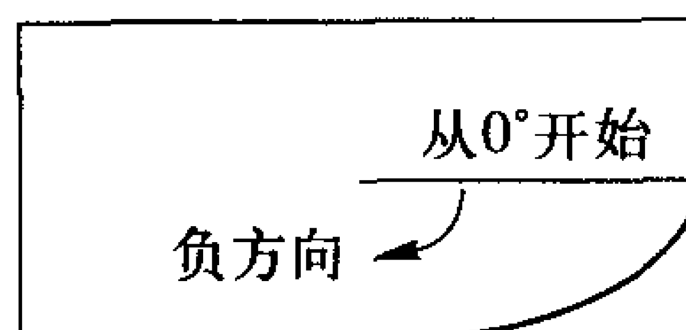
图1-9说明了最后这两个实参如何将不可见椭圆上的一段弧线指定为可见的。倒数第2个实参指定了起始角度。最后一个实参指定了要将椭圆上多少度的弧线变成可见的。如果最后一个实参是360（度），那么，整个椭圆就都是可见的了。

如图1-9所示，角度是从0°开始的。在第一个图形中，起始角度为0°，因此90°的起始角将

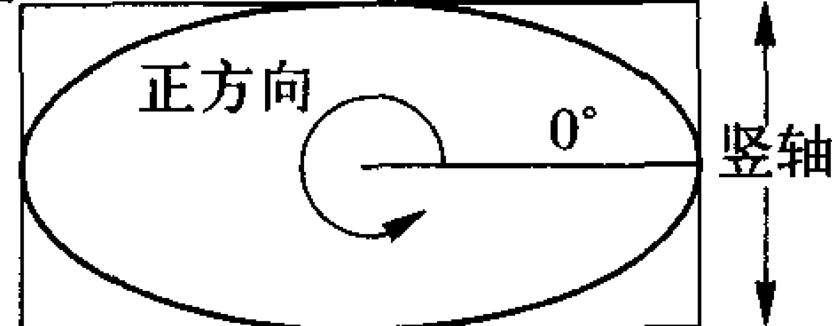
canvas. drawArc(x, y, width, height, 0, 90);



canvas. drawArc(x, y, width, height, 0, -90);



canvas. drawArc(x, y, width, height, 0, 360);



canvas. drawArc(x, y, width, height, 180, 90);

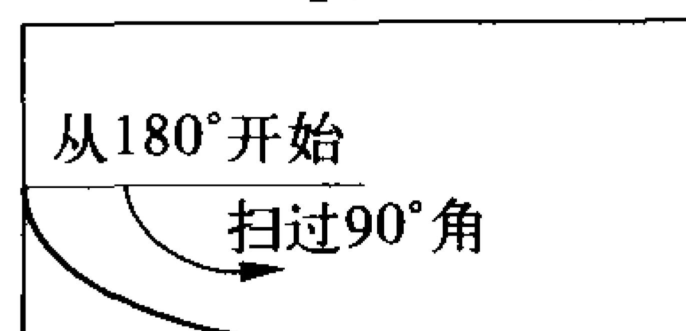


图1-9　指定一段弧线

从椭圆的顶部开始。－90°的起始角将从椭圆的底部开始。逆时针方向是正的。例如，图1-6中笑脸上的笑容起始角为180°，因此它是从不可见椭圆的左端开始的。最后一个实参也是180°，因此，逆时针方向180°的弧线，或者逆时针方向的半个椭圆，成为可见的了。

1.4.5 运行applet

applet是设计为从Web站点上运行的，但要想运行一个applet，并不需要了解如何将其嵌入到Web站点中去。有很多方式可以直接运行applet。一种方式是使用**applet viewer**。用applet viewer运行applet的最简单方法就是在IDE中运行applet，比如本章前面讨论过的Sun公司的ONE Studio（也称为Forte for Java）或TextPad环境。（在TextPad环境中，使用的是`Tools`菜单中的`Run Java Applet`命令。如果弹出一个窗口要求选择一个文件，要回答“No”。这条环境命令会自动激活一个applet viewer。）如果没有IDE，或者无法找到正确的命令来启动applet viewer，请向专家咨询。

结束一个applet的方式取决于你是如何运行这个applet的。如果使用的是applet viewer（或者是从一个环境中运行的applet），则可以单击关闭窗口按钮来结束applet的显示。关闭窗口按钮的位置很可能如图1-6所示，但也可能在别的位置，这是由所用的计算机和操作系统决定的。如果关闭窗口按钮的位置与图1-6显示的不同，它很可能和所用计算机上其他窗口的关闭窗口按钮所处的位置相同。如果是从Web站点上运行applet，这个applet会持续运行到关闭了它所在的页面或者从它所在的页面跳转为止。

快速参考：drawArc

方法`drawArc`会画出一条弧线，即画出椭圆的一部分。

语法：

canvas.drawArc(*x*, *y*, *Width*, *Height*, *StartAngle*, *DegreesShown*);

画出横轴为*Width*像素、竖轴为*Height*像素的椭圆的一部分。放置椭圆时使紧紧包围椭圆的长方形的左上角位于点（*x*，*y*）。所画的弧线区域由*StartAngle*和*DegreesShown*给出，参见图1-9，以获取更详细的信息。

举例：

```
canvas.drawArc(150,160,100,50,180,90);
```

常见问题：什么是canvas？

在下列（从图1-6中复制的）`paint`方法的定义中，标识符`canvas`命名的是什么？

```
public void paint(Graphics canvas)
{
    canvas.drawOval(100,50,200,200);
    canvas.fillOval(155,100,10,20);
    canvas.fillOval(230,100,10,20);
    canvas.drawArc(150,160,100,50,180,180);
}
```

标识符`canvas`命名的是一个进行绘画的对象。最后会完整解释它，现在只要注意到`canvas`是一

个Java用来进行绘画的“哑变量”即可。不一定非要用标识符canvas，但使用时要保持一致。例如，如果把其中一个canvas改成pen，就必须将所有的canvas都改成pen。

```
public void paint(Graphics pen)
{
    pen.drawOval(100,50,200,200);
    pen.fillOval(155,100,10,20);
    pen.fillOval(230,100,10,20);
    pen.drawArc(150,160,100,50,180,180);
}
```

上面的两种paint方法定义是等效的。本章和第3章一直都会使用标识符canvas。

自测题

29. 如何修改图1-6中的applet程序才能使眼睛变成圆的，而不是椭圆的呢？
30. 如何修改图1-6中的applet程序才能让脸变得不高兴呢？（提示：将笑容上下颠倒一下）。

小　结

- 计算机的主存储器中装载了当前正在执行的程序，还装载了很多程序正在操作的数据项。计算机的主存储器被划分成一系列被称为字节的、编了号的单元。
- 计算机的辅助存储器基本上以永久方式来保存数据。
- 编译器是一个程序，它能够将用Java这样的高级语言编写的程序翻译成用低级语言编写的程序。Java编译器将Java程序翻译成使用字节码语言的程序。当你给出运行Java程序的命令时，这个字节码程序就被翻译成了机器语言指令，并由计算机来执行这些机器语言指令。
- 面向对象编程的3个主要原则是封装、多态性和继承。
- 算法是一个用来解决问题的指令集。要想成为一个合格的算法，这些指令的表述必须非常完整和精确，使人只需遵循这些指令，而无需填充任何细节，或者做任何指令中没有完全指定的决定。
- 对象是一种有数据与其相关，并可以执行某些动作的程序构造。对象执行的动作称为方法。类定义了对象的类型。同一个类的所有对象都具有相同的方法。
- 在Java语言中，写下对象名，后面跟上一个下圆点（称为点）、方法名，最后跟上在圆括号中的实参，就可以进行方法调用了。
- 可以编写在计算机屏幕上显示图像的applet。①

✓自测题答案

1. 主存储器和辅助存储器。
2. 软件只是程序的另一个名字。
3. 要相加的两个数字。
4. 在这门课程中参加的所有考试的成绩。
5. 机器语言程序是以一种计算机可以直接执行的方式编写的。高级语言程序是以一种更易于人类读写的方式编写的。在计算机执行高级语言程序之前，必须将其翻译成机器语言程序。Java字节码是一种低级语言，类似于大部分常见计算机的机器语言。将Java字节码表示的程序翻译成几乎所有计算机使用的机器语言是相对容易的。

① 1.4节中介绍的内容。

6. Java是一种高级语言。
7. Java字节码是一种低级语言。
8. 编译器会将高级语言程序翻译成机器语言程序或Java字节码程序这样的低级语言程序。编译Java程序时，编译器会将Java程序翻译成用Java字节码表示的程序。
9. 输入到编译器中的高级语言程序称为源程序。
10. 将Java字节码指令翻译成机器语言指令的程序被称为解释器，通常也称为Java虚拟机（JVM）。
11. 方法是对象可以执行的动作。（在其他编程语言中，方法被称为函数或过程。）
12. 类是对象的种类。同一个类的所有对象都具有相同类型的数据和相同的方法。
13. 是的，同一个类的所有对象都具有相同的方法。
14. 封装是将对象中所有与了解如何使用对象没有必然关系的细节都隐藏起来的过程。换句话说，封装是一种描述类或对象的过程，它只为程序员提供了使用类或对象所必需的信息。
15. “信息隐藏”是“封装”的另一种说法。
16. 在Java这样的编程语言中，多态性意味着，作为指令使用的方法名可以根据执行动作的对象类型来引发不同的动作。
17. 算法是一个用来解决问题的指令集。要想成为一个合格的算法，这些指令的表述必须非常完整和精确，使人只需遵循这些指令，而无需填充任何细节或者做任何指令中没有完全指定的决定。
18. 语法错误是程序中语法方面的错误。对于如何编写程序，有着非常严格的语法规则，如果违反了其中的一条规则（比如，忽略了一个必需的标点符号）就会产生语法错误。
19. 逻辑错误是程序算法中存在的概念性错误。如果程序可以运行并能产生输出，但输出不正确，就说明程序中存在逻辑错误。
20. 语法错误和运行时错误。
21. 是一个逻辑错误。
22. `Java is great!`
23.
```
System.out.println("Java for one.");
System.out.println("Java for all.");
```
24. `mary.increaseAge(5);`
25. 这条语句读入了键盘上输入的一个整数，并将其存储在变量n1中。
26.
```
public class ExerciseProgram
{
    public static void main(String[] args)
    {
        System.out.println("Hello world!");
    }
}
```
在编写的程序中，有些细节（比如标识符的名字）可以与之不同。但一定要编译并运行所编写的程序。
27. 包含了名为SuperClass的类的文件应该叫作SuperClass.java。
28. SuperClass.class.
29. 将代码行
```
canvas.fillOval(155, 100, 10, 20);
canvas.fillOval(230, 100, 10, 20);
```
改成

```
canvas.fillOval(155, 100, 10, 10);
canvas.fillOval(230, 100, 10, 10);
```

将每行最后两个数字从10,20改成了10,10。可以用其他数字，比如20,20，来取代10,10。

30. 将代码行

```
canvas.drawArc(150, 160, 100, 50, 180, 180);
```

改成

```
canvas.drawArc(150, 160, 100, 50, 180, -180);
```

将最后一个数字从正值改成了负值。正确的答案不止一个。例如，下面的答案也是对的：

```
canvas.drawArc(150, 160, 100, 50, 0, 180);
```

将上面任意一个答案中的第一个数字150改大一点也可以。还会有其他正确答案，但都与上述答案类似。

● 编程项目

1. 获取图1-5中显示的Java程序副本（在前言中给出的Web站点上有这个程序）。将文件命名为FirstProgram.java。编译程序，使其不再产生编译器错误信息。然后，运行FirstProgram.java中的程序。
2. 修改在编程项目1中编写的Java程序，使其将3个而不是2个数字相加。对程序进行编译，使其不再产生编译器错误消息，然后运行程序。
3. （要完成这个项目，需要学习1.4节。）编写一个与图1-6中的程序类似的、显示雪人图片的applet程序。（提示：画3个圆，一个在另一个的上面。从下到上，圆圈越来越小。把最上面的那个圆做成一张笑脸。）

第2章

基本类型、字符串及控制台I/O

primitive（形容词）1. 不是从其他东西中导出的；初级的或基本的。

string（名词）1. 绳子，通常由纤维制成，用于紧固、打结或系带…… 6.计算机科学。被计算机当作一个单独条目处理的一系列连续字符。

——《美国传统英语字典》（第3版）

本章将详细介绍Java语言，通过这些介绍就可以编写简单的Java程序了。理解本章的内容不需要具有任何编程经验。如果熟悉其他编程语言（如C、C++、Pascal、BASIC或FORTRAN），那么一定都很熟悉2.1节的很多内容。但是，即使了解这些概念，也应该学习一下Java表达这些概念的方式。

目标

- 熟悉Java用于数字、字符及类似的简单数据的数据类型。这些类型被称为基本类型。
- 了解赋值语句及表达式。
- 弄清楚Java用于字符串的数据类型，学习如何进行简单的字符串处理。这些内容还有助于熟悉类、方法和对象的表示法。
- 了解简单的键盘输入和屏幕输出。
- 选读，了解如何使用`JOptionPane`类实现基于视窗的输入和输出。

预备知识

如果没有读过第1章，至少应该读一下1.3.3节，以熟悉类、方法和对象的表示法。

在阅读本章之前，不需要阅读过1.4节。除了一个例子之外，甚至不需要在阅读2.5节之前阅读1.4节。这个例子就是2.5节的编程示例“应用于图形applet的样式规则”，它用到了1.4节中的内容。但是，如果你没有学过1.4节，可以跳过这个编程示例。

2.5节讲述了`JOptionPane`，独立于其他章的“图形编程补充”小节。要学习后续各章的“图形编程补充”小节，并不需要学习2.5节。

2.1 基本类型和标识符

一旦理解了编程中使用变量的方式，就理解了编程的精华。

——E. W. Dijkstra，图灵奖得主，*Notes on Structured Programming*

本节将介绍如何在Java程序中使用变量和算术表达式。

2.1.1 变量

程序中的变量（variable）是用来存储数字和字母这样的数据的，可以视为某种类型的容器。变量中的数字、字母或其他数据项称为它的值（value）。值是可以改变的，因此，比如，有时候变量的值是6，程序运行一段时间之后，变量中包含的可能会是一个不同的值，比如4。在图2-1给出的程序中，numberOfBaskets、eggsPerBasket和totalEggs都是变量。例如，在运行这个程序时，下列语句会将eggsPerBasket的值设为6：

```
eggsPerBasket = 6;
```

稍后，当程序执行下列语句时，变量eggsPerBasket的值改成了4：

```
eggsPerBasket = eggsPerBasket - 2;
```

下面会对这两条语句进行更详细的解释。

```
public class EggBasket
{
    public static void main(String[] args)
    {
        int numberOfBaskets, eggsPerBasket, totalEggs;

        eggsPerBasket = 6;
        numberOfBaskets = 10;

        totalEggs = numberOfBaskets * eggsPerBasket;

        System.out.println("If you have");
        System.out.println(eggsPerBasket + " eggs per basket and");
        System.out.println(numberOfBaskets + " baskets then,");
        System.out.println("the total number of eggs is " + totalEggs);

        System.out.println("Now we take two eggs out of each basket.");

        eggsPerBasket = eggsPerBasket - 2;
        totalEggs = numberOfBaskets * eggsPerBasket;

        System.out.println("You now have");
        System.out.println(eggsPerBasket + " eggs per basket and");
        System.out.println("and " + numberOfBaskets + " baskets.");
        System.out.println("The new total number of eggs is "
                                                     + totalEggs);
    }
}
```

屏幕对话示例

```
If you have
6 eggs per basket and
10 baskets, then
the total number of eggs is 60
Now we take two eggs out of each basket.
You now have
4 eggs per basket and
10 baskets.
The new total number of gees is 40
```

图2-1 一个简单的Java程序

在Java中，变量是作为存储单元实现的（第1章中讨论过存储单元的概念）。每个变量都分配了一个存储单元。在为变量赋值时，该值（被编码成0、1串）就被放置在变量的存储单元中。

变量的命名规则遵循2.1.2节给出的Java标识符拼写规则。除此之外，还应该选择一些有帮助的变量名。变量名应该能让人想到变量的用途，或者说明它们要装载的数据类型。例如，如果变量是用来对某些东西计数的，可以将其称为count。如果变量是用来存放汽车速度的，可以将变量称为speed。最好永远都不要使用x和y这样的单字母变量名。阅读下列代码的人不会知道程序实际上是在对什么进行加法操作：

```
x = y + z;
```

要运行程序，计算机必须了解程序中每个变量的一些基本信息。它需要知道变量的名字、要为变量保留多大的计算机存储空间，以及如何将变量中的数据项编码成0、1串。只要编译器知道了变量名和变量中存储的数据类型（这样最终计算机也就会知道了），就可以获知所有这些信息了。可以通过声明（declare）变量给出这些信息。使用Java程序中的每个变量之前都必须先对其进行声明。例如，下列来自图2-1的代码行将numberOfBaskets、eggsPerBasket和totalEggs声明为int类型的变量：

```
int numberOfBaskets, eggsPerBasket, totalEggs;
```

变量声明包括一个类型名，后面跟着一个由逗号分隔的变量名列表。声明以分号结束。列表中命名的所有变量都被声明为声明开头给出的类型。

int类型是最常用的整数变量类型，如42、-99、0和2001。int是integer（整数）的缩写。

变量的**类型**（type）决定了变量可以赋什么样的值。如果类型为int，变量就可以赋整数值；如果类型为double，变量就可以赋带有小数点而且小数点后面有尾数部分的数值；如果类型为char，变量就可以赋值为来自计算机键盘的任意字符了。

变量声明告诉计算机变量中会赋什么类型的数据。不同的数据类型是以不同的方式存储在计算机存储器中的，因此，为了了解如何向计算机存储器中存储变量的值，以及如何从计算机存储器中将变量的值恢复出来，计算机必须要知道变量的类型。

快速参考：变量声明

在Java程序中，在使用变量之前必须对其进行声明。变量是按如下方式声明的：

语法：

Type Variable_1, Variable_2, ⋯ ;

举例：

```
int styleNumber, numberOfChecks, numberOfDeposits;
char answer;
double amount, interestRate;
```

Java中有两种主要的类型：类类型和基本类型。如其名字所示，**类类型**（class type）是一个类的类型，即带有数据和方法的对象的类型。**基本类型**（primitive type）简单一些。基本类型的值是简单的和不可分解的，如单个数字或单个字母。“If you have”这样的带引号的字符串是类类型String的值，将在本章稍后对其进行讨论。类型int、double和char是基

本类型。按照惯例，类类型名以大写字母开头，基本类型名以小写字母开头，但类类型和基本类型的变量名都是以小写字母开头的。类类型和基本类型的变量名以相同的方式声明，但是，在类类型变量中存储值的机制与在基本类型变量中存储值的机制不同。在本章和下一章中，我们将主要介绍基本类型。偶尔会使用类类型变量，但只会在它们的行为与基本类型变量的行为非常像的情况下才会使用。将在第4章详细介绍类类型变量。

在使用Java程序中的每个变量之前，都必须对其进行声明。通常，变量的声明放在使用变量之前，或者放在花括号{}包围的程序段的起始处。在到目前为止我们见过的几个简单程序中，这就意味着变量是在变量使用之前，或者在下列代码之后声明的：

```
public static void main(String[] args)
{
```

快速参考：语法

正确编写一个程序或部分程序的规则（一种编程语言的语法规则）称为语言的**语法**。

在本书的很多方框中都是用**语法变量**（syntactic variable）来描述Java语法的。例如，在本章前面名为“变量声明”的方框中，我们使用了下列描述：

语法：

Type Variable_1, Variable_2, …;

单词*Type*、*Variable_1*和*Variable_2*都是语法变量的实例，它们是用一种不同的字体书写的，并使用了下划线符号，这样就可以很容易地将它们识别出来。语法变量指的并不是Java程序中可能出现的单词。相反，它们是一类需要用（一个或多个）合适的Java单词来填充的空白。可以用任意Java类型来取代*Type*（类型）。例如，可以用`int`、`double`、`char`或任何其他类名来取代*Type*。可以用任意变量名来取代*Variable_1*和*Variable_2*。例如，可以用`styleNumber`来取代*Variable_1*，用`numberOfChecks`来取代*Variable_2*，…说明变量列表可以是任意长的。例如，可以用

```
int styleNumber, numberOfChecks, numberOfDeposits;
```

取代

Type Variable_1, Variable_2, …;

以获取可以在Java程序中使用的合法变量声明。在本书中，为了有助于理解这些语法表达式，通常会在它们后面跟上一个或多个用这种语法表达式说明的Java实例。

记住：语法变量

在本书中看到*Type*、*Variable_1*或*Variable_2*这样的单词时，要记住，它们并不会直接出现在Java代码中。它们是语法变量，这就意味着可以用它们所描述的类别中的某个类型来取代它们。例如，可以用`int`、`double`、`char`或任意其他类型名来取代*Type*，可以用任意变量名来取代*Variable_1*和*Variable_2*。

2.1.2 Java标识符

在编程语言中，为变量名这样的名字使用的技术术语是**标识符**（identifier）。在Java中，标识符（名字）只能包含字母、数字（0~9）和下划线（_），但名字中第一个字符不能是

数字[1]。尤其要注意，名字中不能包含空格或任何其他字符，比如点号（.）或星号（*）。名字的长度没有限制（实际上长度总是有限的，但对此没有官方的限制，Java甚至可以接受那些长得出奇的名字）。Java是**区分大小写的**。这就意味着大写和小写字母会被当作不同的字符。例如，Java认为mystuff、myStuff和MyStuff是3个不同的名字，可以用这3个名字表示3个不同的变量（或其他项）。当然，使用两个仅在大小写方面有所区别的名字是不好的编程习惯，但Java编译器会很乐意接受。在这些规则规定的范围之内，可以为Java程序中定义的变量或类，或者任意其他项使用你想用的任何名字。但在选择名字时还是有一些样式指导原则的。

下面解释一下大小写字母的一些特殊用法，如numberOfBaskets。用NumberOfBaskets或number_of_baskets取代numberOfBaskets是完全合法的，但这两个名字违反了一些关于如何使用大小写字母且得到广泛认可的惯例。根据惯例，只用字母和数字来书写名字。因为不能使用空格，所以通常用大写字母来“断开”由多个单词组成的名字。下面列出的都是合法的名字，这些名字也遵循了上述惯例。

```
inputStream YourClass CarWash hotCar theTimeOfDay
```

下列名字在Java中都是不合法的，使用其中任意一个，编译器都会报警：

```
My.class netscape.com go-team 7eleven
```

前3个名字中都含有非法字符：一个点或一个连字符。最后一个名字是由数字开头的，因此也是非法的。

注意，有些合法的名字是以大写字母开头的，而其他一些，如hotCar，则是以小写字母开头的。通常都要遵循一个惯例，即类的名字以大写字母开头，而变量、对象和方法的名字以小写字母开头。

当然，在Java程序中有一些单词，如if，不是用来命名变量、类或对象的。if这样的单词称为**关键字**（keyword）或**保留字**（reserved word）。在Java语言中，这些关键字具有特殊的、预先定义好的含义，不能将其用作类、对象或除其原义之外的任何事物的名字。附录A给出了Java关键字的完整列表，但通过应用来学习这些关键字会更容易一些。一种特殊的颜色来表示。其他一些单词，如main和println，具有预先定义好的含义，但不是关键字。这就意味着可以改变它们的含义，但这样容易把你自己或其他阅读你的程序的人搞糊涂，所以这不是什么好主意。

快速参考：名字（标识符）

Java程序中某个项的名字，如变量、类、方法或对象的名字，一定不能以数字开头，并且只能包含字母、数字（0~9）和下划线（_）。大写字母和小写字母会被当作不同的字符。（也可以使用$符号，但它是被保留作特殊用途的，因此不应该在Java名字中使用$符号。）

程序中的名字通常称为**标识符**。

尽管Java语言没有这样要求，但按照惯例及本书所遵循的规则，类名都是以大写字母开头的，而变量、对象和方法名都以小写字母开头。这些名字通常只由字母和数字组成。

---

[1] Java的确允许标识符中出现美元符号$，但这些标识符是有特殊含义的，因此，不应该在你的标识符中使用$符号。

▲ 易犯错误：Java是区分大小写的

不要忘记Java是区分大小写的。如果使用了一个myNumber标识符，然后在程序的另一部分将标识符拼成MyNumber，Java是不会把它们当作同一个标识符的。要想将它们当作相同的标识符，就必须使用完全相同的写法（大小写相同）。△

2.1.3 基本类型

图2-2给出了所有的Java基本类型。注意，有4种整数类型，即byte、short、int和long。各种不同整数类型之间唯一的区别就是所存储的整数的范围，以及所使用的计算机存储器的容量。如果无法确定应该使用哪种整数类型，就使用int类型。

类型名	值的类型	使用的存储器	容量范围
byte	整数	1字节	−128~127
short	整数	2字节	−32 768~32 767
int	整数	4字节	−2 147 483 648~2 147 483 647
long	整数	8字节	−9 223 372 036 854 775 808~ 9 223 372 036 854 775 807
float	浮点数	4字节	$\pm 3.402\,823\,47 \times 10^{+38}$~$\pm 1.402\,398\,46 \times 10^{-45}$
double	浮点数	8字节	$\pm 1.767\,693\,134\,862\,315\,70 \times 10^{+308}$~ $\pm 4.940\,656\,458\,412\,465\,44 \times 10^{-324}$
char	单字符 (Unicode)	2字节	所有Unicode字符
boolean	true（真） 或false（假）	1位	不适用

图2-2 基本类型

不带小数的数如0、1、−1、2或−2等称为**整数**（integer）。带有小数部分的数，如9.99、3.14159、−5.63或5.0，称为**浮点数**（floating-point number）。注意，5.0是浮点数，而不是整数。如果请求计算机包含一个小数部分，而那个小数部分恰好为0，也不会改变数字的类型。如果它有小数部分，即使小数部分为0，也是浮点数。如图2-2所示，Java有两种浮点数类型：float和double。例如，下列代码声明了两个变量，一个是float类型的，另一个是double类型的：

```
float cost;
double capacity;
```

如果无法确定应该使用float还是double类型，就使用double。

基本类型char用于单个字符，比如字母或百分号。例如，下列语句将变量symbol声明为char类型的，并将大写字符A存储在symbol中，然后将值写到屏幕上，使字符A出现在屏幕上。

```
char symbol;
symbol = 'A';
System.out.println(symbol);
```

注意，在Java程序中给出一个字符时，会将其包围在单引号中，而不是包围在双引号中。在字符的两边使用的是相同的引用符号。那个符号既作为左引号使用又作为右引号使用。要

特别注意，大写字母和小写字母是不同的字符。例如，'a'和'A'是不同的字符。

我们要讨论的最后一个基本类型是boolean类型。boolean类型有两个值，即true（真）和false（假）。这就意味着可以用boolean类型的变量来存储一个是真还是假问题的答案，比如“myTime小于yourTime吗？”。在第3章中，将对boolean类型进行更详细的介绍。

2.1.4 赋值语句

为变量赋值，或者修改变量值的最直接的方法就是使用**赋值语句**（assignment statement）。例如，如果answer是int类型的变量，想为其赋值42，就可以使用下列赋值语句：

```
answer = 42;
```

等号（=）用于赋值语句中时，称为**赋值运算符**（assignment operator）。这并不代表等号在其他上下文中的含义。赋值语句是一条命令，它告诉计算机将存储在赋值运算符左边的变量中的值改成右边表达式的值。因此，一条赋值语句通常包含一个变量，后面跟着赋值运算符（等号），后面再跟一个表达式。赋值语句以分号结束。因此赋值语句采用下列形式：

Variable = Expression;

表达式可以是另一个变量、数字或更复杂的表达式。复杂的表达式可能是用加号（+）和乘号（*）这样的算术运算符将变量和数字组合起来形成的。

例如，下列语句都是赋值运算符的实例：

```
amount = 3.99;
firstInitial = 'B';
score = numberOfCards + handicap;
eggsPerBasket = eggsPerBasket - 2;
```

（所有的名字，如amount、score和numberOfCards，都是变量。假定变量amount是double类型，firstInitial是char类型，其余的变量都是int类型。）

执行一条赋值语句时，计算机首先会对赋值运算符（=）右边的表达式进行计算，以获取表达式的值。然后用该值来设置赋值运算符（=）左边的变量的值。可以认为赋值运算符是在说“让变量的值等于跟在它后面的那个值”。

例如，如果变量numberOfCards的值为7，handicap的值为2，则下列语句会使变量score的值变成9：

```
score = numberOfCards + handicap;
```

图2-1中的下列代码行是另一种赋值语句示例：

```
totalEggs = numberOfBaskets * eggsPerBasket;
```

这条赋值语句告诉计算机设置totalEggs的值，使其等于变量numberOfBaskets中的数值乘以变量eggsPerBasket中的数值。星号（*）在Java中用作乘法符号。

注意，变量可以出现在赋值运算符（=）的两边，并会以一种开始看起来有点奇怪的方式出现。例如：

```
count = count + 10;
```

这并不意味着count的值等于count的值加10，这当然是不可能的。应该说，这条语句告诉计算机在count原来的值上加10，然后将其作为count的新值，这就意味着这条语句会将count的值增加10。记住，在执行赋值语句时，计算机会先计算出赋值运算符右边表达式的值，然

后将所得值作为赋值运算符左边变量的新值。

下面是图2-1中的赋值语句：

```
eggsPerBasket = eggsPerBasket - 2;
```

这条赋值语句会将eggsPerBasket的值减2。

上面赋值语句中的数字2称为**常量**（constant）。它与变量eggsPerBasket不同，2的值是无法改变的，所以称为常量（常量有时也被称为文字常量（literal））。常量不一定是数字。字符'A'、'B'和'$'就是3个char类型的常量。它们的值无法改变，但可以在赋值语句中用它们来修改char类型变量的值。例如，下列语句将变量firstInitial的值改成了'B'：

```
firstInitial = 'B';
```

在这条赋值语句中，变量firstInitial在通常情况下都会是char类型的。

类似地，下列语句将变量price的值改成了9.99：

```
price = 9.99;
```

在这条赋值语句中，变量price在通常情况下都会是double类型的（尽管它也可以是float类型的）。它不可能是int类型或char类型。就像有句格言说的："你不能把一个方钉子钉到一个圆洞里去"，同样，也不能把一个double类型的值放到int类型的变量中去。

快速参考：基本类型的赋值语句

一条等号左边为基本类型变量的赋值语句会引发下列动作：首先，对等号右边的表达式进行计算，然后，用所得值来设置等号左边变量的值。

语法：

Variable = *Expression*;

举例：

```
score = goals - errors;
interest = rate * balance;
number = number + 5;
```

2.1.5 特殊的赋值运算符

可以将简单的赋值运算符（=）与加号（+）等算术运算符结合起来，生成一种具有特殊目的的赋值运算符。例如，下列语句会将变量amount的值加5。

```
amount += 5;
```

这条语句实际上只是下列语句的缩写。

```
amount = amount + 5;
```

这不是什么了不起的事，但有时候用起来会很方便。

可以用任何其他的算术运算符，如-、*、/和%，来完成同样的操作（2.1.11节将介绍%运算符）。例如：

```
amount = amount * 25;
```

可以用下列等价的代码行来取代这一行：

```
amount *= 25;
```

2.1.6 简单的屏幕输出

现在我们将对屏幕输出进行简单的介绍——只是让你能够编写并理解图2-1那样的程序。2.3节会继续讨论屏幕输出。

正如在第1章中提到的，System.out是一个对象，println是这个对象的一个方法，它将输出发送到屏幕。因此

```
System.out.println(eggsPerBasket + " eggs per basket.");
```

会将变量eggsPerBasket的值输出到屏幕，并在后面跟上短语"eggs per basket"。注意，加号在这里不是用来做算术运算的，它表示另一种类型的"与"。可以将前面的输出语句当作一条指令，这条指令输出了变量eggsPerBasket的值，并接着输出了字符串"eggs per basket"。

2.1.7 简单的输入

在图2-1中，用赋值语句设置变量的值。而从用户那里获取计算所需的数字会更有意义，这样就可以再次以不同的数字运行程序了。在图2-3中，重新编写了图2-1中的程序，使用户可以从键盘输入数字。2.3节会详细介绍这种键盘输入。本小节所介绍的知识，足够使你在2.3节之前，理解在程序中进行的简单键盘输入。

```
import java.util.*;  <——— 包含Scanner类的包（库）的名字。

public class EggBasket2
{
    public static void main(String[] args)
    {
        int numberOfBaskets, eggsPerBasket, totalEggs;

        Scanner keyboard = new Scanner(System.in);  <— 进行设置，使程序可以接受键盘输入。
        System.out.println("Enter the number of eggs in each basket:");
        eggsPerBasket = keyboard.nextInt();  <——— 从键盘读入一个整数。
        System.out.println("Enter the number of baskets:");
        numberOfBaskets = keyboard.nextInt();

        totalEggs = numberOfBaskets * eggsPerBasket;

        System.out.println("If you have");
        System.out.println(eggsPerBasket + " eggs per basket and");
        System.out.println(numberOfBaskets + " baskets, then");
        System.out.println("the total number of eggs is " + totalEggs);

        System.out.println("Now we take two eggs out of each basket.");

        eggsPerBasket = eggsPerBasket - 2;
        totalEggs = numberOfBaskets * eggsPerBasket;

        System.out.println("You now have");
        System.out.println(eggsPerBasket + " eggs per basket and");
        System.out.println(numberOfBaskets + " baskets.");
        System.out.println("The new total number of eggs is "
                                            + totalEggs);
    }
}
```

图2-3 带有键盘输入的程序

屏幕对话示例

```
Enter the number of eggs in each basket:
6
Enter the number of baskets:
10
If you have
6 eggs per basket and
10 baskets, then
the total number of eggs is 60
Now we take two eggs out of each basket.
You now have
4 eggs per basket and
10 baskets.
The new total number of eggs if 40
```

图2-3　（续）

为了用现在使用的方式实现键盘输入，你的程序文件必须在文件起始处包含下列代码：

```
import java.util.*;
```

这行代码告诉Java编译器到哪里去找用于键盘输入的Scanner类的定义。

下列代码行对变量进行了设置，以便从键盘输入数据：

```
Scanner keyboard = new Scanner(System.in);
```

这行代码和赋值语句以及其他程序指令，都是程序主体的一部分。这行代码必须出现在第一条从键盘获取输入的代码行之前。

第一条获取键盘输入的代码行是：

```
eggsPerBasket = keyboard.nextInt();
```

这是一条赋值语句，它为变量eggsPerBasket赋值。等号右边的表达式，即

```
keyboard.nextInt()
```

从键盘读入了一个int值。赋值语句将这个int值赋给了变量eggsPerBasket，成为eggsPerBasket的新值。

从键盘输入数字时，用户必须将数字放在单独一行，或者用一个或多个空格来区分多个数字。

2.1.8 数字常量

变量的值是可以改变的。这就是它被称为变量的原因：它的值是*可变的*。类似2这样的文字数值是无法改变的，它总是2，永远也不会是3。2或3.7这样的文字值称为**常量**（constant），因为它们的值不会改变。除数字类型之外的其他类型的文字表达式也称为常量。例如'Y'就是char类型的常量。本质上只能用一种方法来书写char类型的常量，即将字符放在单引号内。数字常量的书写规则要更复杂一些。

整数类型常量是以你预期的书写方式书写的，比如2、3、0、-3或752。整数常量的前面可以带有一个加号或减号，如+12和-72。数字常量中不能包含逗号。在Java中，数字表达式1,000是不正确的。整数常量中不能包含小数点，带有小数点的数字是浮点数。

可以用两种形式中的任意一种来书写浮点常量。一种简单的形式是像日常书写数字那样，

在小数点后面跟上一些数字。另一种稍微复杂一点的形式与常用的记数法类似。

浮点常量较复杂的记法通常被称为**e记数法**（e notation）、**科学记数法**（scientific notation）或者**浮点记数法**（floating-point notation）。例如，对于数字865000000.0，可以用下列记法以一种更清晰的方式来表示，数学和物理学中使用了这种记法，但Java中没有用

8.65×10^8

Java有一种类似的表示法，但是由于键盘上无法输入指数，就将10省略了，用字母e取代了乘号和10。因此，在Java中，将8.65×10^8写成8.65e8（或者将其写成一种不太方便的形式865000000.0）。在Java程序中，8.65e8和865000000.0这两种形式是等价的。

类似地，数字4.83×10^{-4}等于0.000483，在Java中可以写作4.83e-4。e表示的是指数（exponent），因为e的后面跟着一个被当作10的指数的数字。

因为被10乘的效果和在数字中移动小数点的效果是相同的，所以，可以认为e后面的数字是说明将小数点向右移动多少位。如果e后面的数字是负数，就将小数点向左移动多少位。例如，2.48e4和24800.0是等价的，2.48e-2和0.0248是等价的。

e前面的数字可以是一个带有小数点的数字，也可以是一个不带小数点的数字。e后面的数字中不能包含小数点。

常见问题：浮点数中的“浮”是什么意思？

根据本小节对e记数法的描述，可以通过调整指数将小数点“浮动”到一个新的位置，浮点数就是据此得名的。用4.83e-4来表示0.000483，就可以将这个数字的小数点浮动到4的后面。计算机语言的实现者们通过这种技巧，将每个浮点数都作为一个小数点前面只有一位数字（并带有适当指数）的数字来存储。由于实现过程中经常会对这些数字中的小数点进行浮动，因此，将它们称为浮点数。（实际上，数字是作为以2为底的数存储，而不是像我们在示例中使用的以10为底的数存储的，但它们的原理是相同的。）

2.1.9 赋值兼容性

正如前面提到的那样，试图把一种类型的值赋给另一种类型的变量就像试图将一个方形的钉子钉到一个圆形的洞里面去一样。不能将一个42这样的int值放到一个char类型的变量中；也不能将3.5这样的double值放到int类型的变量中；你甚至不能将double值3.0放到int类型的变量中。除非通过某种方式对值进行转换，使其与变量类型相匹配，否则就无法将一种类型的值存储到另一种类型的变量中。但是，在处理数字时，有时（但不总是）会自动执行这种转换。将一个整数类型的值赋给一个浮点类型的变量时，会执行这种转换，例如：

```
double doubleVariable;
doubleVariable = 7;
```

下面这些赋值语句也会自动地执行转换：

```
int intVariable;
intVariable = 7;
double doubleVariable;
doubleVariable = intVariable;
```

一般来说，可以将下面的列表中任意一种类型的值赋给沿着箭头方向出现在它后面的任意类

型的变量：

```
byte --> short --> int --> long --> float --> double
```

例如，可以将一个long类型的值赋给float类型，或double类型的变量（当然也可以赋给long类型的变量），但是不能将long类型的值赋给byte、short或int类型的变量。（注意，这并不是任意列出的类型序列。在列表上从左向右，类型所允许的值范围越来越大，还可以允许在数字中使用小数点，因此类型就变得越来越复杂了。）

可以将一个char类型的值赋给一个int类型的变量，或者赋给类型列表中跟在int后面的任意数字类型的变量。在讨论键盘输入时，这种特别的赋值兼容性是很重要的。但是，除非在某些特殊的情况下，否则我们并不建议将char类型的值赋给int类型的变量[①]。

如果想将一个double类型的值赋给int类型的变量，就必须像我们接下来要解释的那样，通过强制类型转换来改变值的类型。

快速参考：赋值兼容性

可以将下面列表中任意类型的值赋给沿着箭头方向出现在其后面的任意类型的变量：

```
byte --> short --> int --> long --> float --> double
```

尤其要注意的是，可以将任意一种整数类型的值赋给任意一种浮点类型的变量。

将char类型的值赋给int类型的变量，或者赋给类型列表中任意一种跟在int后面的数字类型变量，都是合法的。

2.1.10 强制类型转换

在Java及大部分编程语言中，**强制类型转换**（type cast）是将值的类型从它通常的类型转换成其他类型。例如，将2.0的类型从double转换成int类型。在2.1.9节中，我们描述了何时可以将一种类型的值赋给另一种类型的变量，并自动执行类型转换。在其他情况下，如果想将一种类型的值赋给另一种类型的变量，就必须执行强制类型转换。下面看看在Java中这是如何实现的。

假设有下列代码：

```
double distance;
distance = 9.0;
int points;
points = distance;    // 这是非法的赋值。
```

正如注释指出的那样，最后一条语句在Java中是非法的。即使double类型值的小数点后面恰好为全0，并因此在概念上是一个整数时，也无法将一个double类型的值赋给一个int类型的变量。

为了将一个double类型的值赋给一个int类型的变量，必须在那个值或者装有那个值的变量前面加“int”。例如，可以用下列语句取代前面的非法赋值语句，从而得到一个合法

① 如果用户使用过其他语言、比如C或C++，在得知无法将char类型的值赋给byte类型的变量时，可能会感到很惊讶。这是因为Java使用的是Unicode字符集而不是ASCII字符集，因此，Java会为每个char类型的值保留两个字节的存储器，而很自然地只为byte类型的值保留一个字节的存储器。这是能使你注意到Java使用了Unicode字符集的几种情形之一。但是，如果你从int转换到char或者进行相反的转换，会得到与平常一样的ASCII数字和字符。

的赋值。

```
points = (int)distance;  ←—— 这是合法的赋值。
```

表达式(int)distance称为强制类型转换。这样并没有改变变量distance中存储的值，但它确实改变了表达式返回的值。因此，在赋值语句

```
points = (int)distance;
```

中，distance和存储在distance中的值都没有任何改变。但是存储在points中的值为存储在distance中的值的“int版本”。如果distance的值为9.0，那么distance的值仍然为9.0，但赋给points的值为9。

要注意，强制类型转换并没有改变(int)distance这样的表达式中源变量的值，这是很重要的。类似(int)25.36或(int)distance这样的表达式是可以产生int值的表达式。因此，如果distance的值为25.36，那么(int)distance的值就是25，但是distance的值仍然是25.36。这种情况与计算一定数量的钱款中的（整数）美元数的情况类似。如果你有25.36美元，那么你拥有的美元数就是25，但25.36美元并没有改变；只是用它来产生了一个整数25。例如：

```
double dinnerBill;
dinnerBill = 25.36;
int dinnerBillPlusTip = (int)dinnerBill + 5;
System.out.println("The value of dinnerBillPlusTip is " + dinnerBillPlusTip);
```

表达式(int)dinnerBill产生了值25，因此，这段代码的输出应该为

```
The value of dinnerBillPlusTip is 30
```

但是变量dinnerBill包含的值仍然是25.36。

一定要注意，在执行从double到int的（或者从任意浮点类型到任意整数类型的）强制类型转换时，数值中小数点后面的部分就被丢弃了，而不会被舍入。这种方式称为截断(truncating)。例如，考虑下面的语句：

```
double dinnerBill;
dinnerBill = 26.99;
int numberOfDollars;
numberOfDollars = (int)dinnerBill;
```

这样并没有将numberOfDollars的值设为27。它将numberOfDollars的值设置成了26。结果没有被舍入。

正如我们前面提到的那样，将一个整数值赋给一个浮点类型的变量（比如double类型的变量）时，会自动地通过强制类型转换将整数转换为变量的类型。例如：

```
double point;
point = 7;
```

这条赋值语句等价于

```
point = (double)7;
```

在第一种赋值版本中，强制类型转换(double)是隐式的。

快速参考：强制类型转换

在很多情况下，都不能将一种类型的值存储到另一种类型的变量中。在这些情况下，必须使用**强制类型转换**将值转换成与目标类型“等价”的值。

语法：

(Type_Name)Expression

举例：

```
double guess;
guess =7.8;
int answer;
answer = (int)guess;
```

存储在answer中的值会是7。注意这个值是被截断，而不是舍入的。还要注意到，变量guess没有任何改变。赋值语句只对存储在answer中的值有影响。

■ Java提示：通过强制类型转换将一个字符转换成一个整数

有时Java会将char类型的值作为整数来对待，但将整数的值赋给字符与字符自身的含义是没有任何联系的。例如，下列强制类型转换将输出对应于字符'7'的int值：

```
char symbol;
symbol = '7';
System.out.println((int)symbol);
```

你可能认为前面的代码会向屏幕输出7，但它没有输出7。它输出的是数字55。Java及所有其他编程语言对字符进行了任意的编号，以生成对应于每个字符的整数。在这种情况下，这些数字就没什么特殊之处了；它们只是和字母或加号一样的字符。因此，也没有作任何努力以使这些数字对应于它们原来的值。基本上，只是把所有的字符都写下来，然后按照书写顺序对其进行编号。字符'7'只是碰巧对应于55而已。(这种编号系统称为Unicode系统，本章稍后将进行讨论。如果你听说过ASCII编号系统，那么，对英语字符来说，Unicode系统和ASCII系统是一样的。)

● 编程提示：初始化变量

已经被声明，但还未通过赋值语句（或其他一些方式）进行赋值的变量，称为**未初始化**(uninitialized)。如果变量是个类变量，它就确实是没有值的。如果是基本类型的变量，则可能会有默认值。但如果显式地向变量赋值，就算是简单地重新为其赋一个默认值，程序都会更清晰一些。(经验表明，默认值的细节经常会发生变化，所以不要认为它们的值会永远不变。)

确保不会有未初始化变量的一种简单方法就是在声明中对其进行初始化。如下例所示，只要将声明和赋值语句结合起来就行了：

```
int count = 0;
double taxRate = 0.075;
char grade = 'A';
int balance = 1000, newBalance;
```

注意，可以在声明中对一些变量进行初始化，对另一些变量不进行初始化。

有时编译器会警告你没有初始化变量。在大多数情况下，这确实会是真的。尽管编译器偶尔也会错误地给出这条警告。但在说服编译器相信出了问题的变量已经被初始化之前，编译器是不会对程序进行编译的。为了让编译器满意，即使在使用变量之前会为其赋另一个值，也要在声明变量时对其进行初始化。在这种情况下，你是无法和编译器争论的。

快速参考：将变量声明与赋值结合起来

可以将变量的声明与为变量赋值的赋值语句结合起来。

语法：

Type Variable_1 = Expression_1 , Variable_2 = Expression_2 , …;

举例：

```
int numberSeen = 0, increment = 5;
double height = 12.34, prize = 7.3 + increment;
char answer = 'y';
```

▲ 易犯错误：浮点数的不精确性

浮点数是以有限的精确度存储的，因此，对所有实用目的来说，它都只具有近似的量值。例如，浮点数1.0/3.0等于0.3333333…其中那3个点说明3是要永远延续下去的。计算机是以一种与前面显示的小数表示方式有些类似的形式来存储数字的，但它只能为有限的数字提供存储空间。如果它只能存储小数点后面的10位数字，那么1.0/3.0就会被存储为0.3333333333（没有其他3了）。

因此，1.0/3.0就是作为一个稍小于1/3的数存储的。换句话说，作为1.0/3.0存储起来的值只是约等于1/3。现实中，计算机是以二进制表示法而不是用以10为底的形式来存储数字的，但它们的原理相同，因此也会发生同样的情况。在将浮点数存储到计算机的过程中，有些浮点数会失去其精确性。

浮点数（如double类型的数）和整数（比如int类型的数）是以不同的方式存储的。正如我们在上一段指出的那样，浮点数实际上是作为近似的量值存储的。另一方面，整数则是作为精确的量值存储的。这种区别有时是很微妙的。例如，从概念上讲，数字5和5.0是相同的数字。但Java认为它们是不同的。整数5是int类型的，具有精确的量值。而数字5.0中包含小数部分（即使小数为0），因此它是double类型的，这样，5.0就是作为仅具有有限精度的值存储的。 △

自测题

1. 下列哪些可以用作Java中的变量名？

   rate1、1stPlayer、myprogram.java、long、TimeLimit、numberOfWindows

2. 一个Java程序中可以有两个名为aVariable和avariable的不同变量吗？
3. 给出对一个名为count的int类型变量的声明。在声明中应将这个变量初始化为0。
4. 给出两个double类型变量的声明。将变量命名为rate和time。在声明中应将这两个变量都初始化为0。
5. 写出对两个名为miles和flowRate的变量的声明。将miles声明为int类型的变量，并在声明中将其初始化为0。将flowRate声明为double类型的变量，并在声明中将其初始化为50.56。
6. 编写一条Java赋值语句，将变量interest的值设置为变量balance的值乘以0.05。
7. 编写一条Java赋值语句，将变量interest的值设置为变量balance的值乘以变量rate的值。这些变量都是double类型的。
8. 编写一条Java赋值语句，将变量count的值加3。变量是int类型的。
9. 下列程序代码行会产生什么样的输出？

```
char a,b;
a = 'b';
System.out.println(a);
b = 'c';
System.out.println(b);
a = b;
System.out.println(a);
```

10. 在本书的Java提示中，我们看到下列代码并没有输出整数7：

```
char symbol;
symbol = '7';
System.out.println((int)symbol);
```

因此，(int)symbol并没有生成与symbol中的数字相对应的数（假定symbol中包含的是'0'、'1'、…、'9'这10个数字中的一个）。你能给出一个表达式，使其生成与symbol中的数字直接对应的整数吗？提示：这些数字确实对应于连续的整数，因此，如果(int)'7'是55，(int)'8'就是56。

11. 下列代码会产生什么样的输出？

```
int result = 10;
result *= 3;
System.out.println("result is " + result);
```

2.1.11 算术运算符

在Java中构建包含加（+）、减（-）、乘（*）、除（/）的算术表达式的方法，与在普通的算术或代数运算中构建算术表达式的方法基本相同。可以用算术运算符+、-、*、/将变量或数字组合起来。这种表达式的含义基本上和你预想的一样，但结果的类型，甚至结果的值偶尔也会有些需要注意的细节。所有的算术运算符都可与任一种整数类型数字、浮点类型的数字，甚至不同类型的数字一同使用。生成值的类型取决于被组合起来的数字的类型。

下面通过一些只组合了两个变量、两个数字、或一个变量和一个数字的简单表达式开始讨论。如果两个操作数（即每个数字或变量）是同一类型的，那么，结果类型不变。如果一个操作数是浮点类型，而另一个操作数是整数类型的，结果是浮点类型的。例如：

```
amount - adjustment
```

如果变量amount和adjustment都是int类型的，结果（返回的值）就是int类型的。如果amount和adjustment中的一个或两者都是double类型的，结果是double类型的。如果用+、*或/中的任意一个运算符取代运算符-，结果的类型也是由同样的方式决定的。

通常可以将使用两个以上操作数的、更长的表达式看作一系列的步骤，每个步骤都只包含两个操作数。例如，要计算表达式

```
balance + (balance * rate)
```

的值，先计算balance*rate，得到一个数字，然后，再用加法把所得到的数字和balance结合起来。这就意味着我们用来确定带有两个操作数的表达式类型的规则同样可以用于更复杂的表达式：如果组合起来的项都是同一种类型的，结果就是那种类型的；如果组合起来的项有些是整数类型的，有些是浮点类型的，结果是浮点类型的。

通常你需要了解的内容就是生成值的类型是整数类型还是浮点类型。但是，如果需要了解一个算术表达式生成值的确切类型，可以按照下列方法确定：生成值的类型是表达式中用到的类型之一。在表达式用到的所有类型中，位于下面序列中（从左向右）最后面的一种类型就是生成值的类型：

```
byte --> short --> int -->long -->float -->double
```

注意，这个序列和用来确定自动类型转换方式的序列是一样的。

除法运算符（/）值得特别关注，因为除法运算中操作数的类型可能会对生成值产生极大的影响。用除法运算符将两个数字结合起来，而且两个数字中至少有一个是double类型（或其他浮点类型）时，结果就是通常你期望的除法运算结果。比如，9.0/2有一个double类型

的操作数，即9.0，因此，结果就是double类型的数4.5。但是，当两个操作数都是整数类型时，结果可能是令人惊讶的。例如，9/2有两个int类型的操作数，因此它会生成int类型的结果4，而不是4.5。小数点后面的尾数部分就这样丢失了。一定要注意，在用两个整数做除法时，结果不会被舍入；无论小数点后面的部分有多长，都会被丢弃（截断）。因此，11/3的结果就是3（而不是3.6666…）。如果小数点后面除了零什么都没有，那么小数点和小数点后面的零也会被丢弃，即使是这种看起来很微小的差别也有很重要的意义。例如，8.0/2计算得到double类型的值4.0，这只是一个近似的量值。但是，8/2计算得到int类型的值4，就是一个精确的量值。4.0的近似特性会影响到任何使用此结果进行的进一步计算的精确性。

将**运算符**%用于整数类型的操作数，可以恢复出与小数点后面的尾数部分等同的内容。用一个整数去除另一个整数时，会得到一个结果（也称为商）和一个余数。例如，14除以4得到3和余数2。换句话说，14除以4得3，还剩下2。%运算符给出了进行除法运算之后的余数，即剩下的量值。因此，由于14除以4得3，还剩下2，所以14/4计算得3，14%4计算得2。

%运算符的应用比较广泛。通过它，程序可以以2、3或任何其他数字为间隔来计数。例如，如果你想每隔一个整数做一件事，就要知道那个整数是偶数还是奇数，这样就能对每个偶数（或者相反，对每个奇数）执行动作了。如果n%2等于0，整数n就是偶数，如果n%2等于1，这个整数n就是奇数。同样，如果希望程序每隔3个整数做一件事，程序可以这样对所有的整数进行遍历：用一个int变量n来存储整数，并对n%3进行测试。在这种情况下，程序只有在n%3等于0时才会执行动作。

2.1.12 圆括号和优先级规则

可以用圆括号来组织算术表达式中的项，方法与代数和算术运算中使用圆括号的方法相同。有了圆括号，你就可以告诉计算机第一步执行哪些操作、第二步执行哪些操作，等等。例如，考虑下面两个只是圆括号位置有所区别的表达式：

```
(cost + tax) * discount
cost + (tax * discount)
```

计算第一个表达式时，计算机首先将cost和tax相加，然后将结果与discount相乘；计算第二个表达式时，计算机首先将tax和discount相乘，然后再将结果与cost相加。如果用一些数字作为变量的值，并执行这两个计算，你会看到它们产生的结果是不同的。

如果省略圆括号，计算机仍然会对表达式进行计算。例如，考虑下列赋值语句：

```
total = cost + tax * discount;
```

这个表达式等效于

```
total = cost + (tax * discount);
```

省略了圆括号时，计算机会在执行加法运算之前执行乘法运算。一般来说，当运算的顺序不是由圆括号确定的时候，计算机会按照图2-4中显示的**优先级规则**（precedence rule）确定的顺序来执行运算。（图2-4显示了本章用到的所有运算符。第3章将给出更多优先级规则，附录B给出了更完整的优先级规则列表。）在列表的上面列出的运算符具有**较高优先级**（higher precedence）。计算机在决定先执行两个运算符中的哪一个，且没有圆括号来指定执行顺序时，会在执行具有较低优先级的运算之前先执行具有较高优先级的运算。有些运算符具有相同的优先级，在这种情况下，运算的顺序就是由运算符从左至右的顺序来决定的。具有相同优先

级的二元运算符是按照从左至右的顺序执行的。具有相同优先级的一元运算符是按照从右至左的顺序执行的。

最高优先级

第一：一元运算符：+、-、++、--和!

第二：二元算术运算符：*、/和%

第三：二元算术运算符：+和-

最低优先级

图2-4　优先级规则

一元运算符（unary operator）是只有一个实参（应用运算符的对象）的运算符，就像下列赋值语句中的运算符-。

```
bankBalance = -cost;
```

二元运算符（binary operator）有两个实参，就像下列语句中的运算符+和*。

```
total = cost + (tax * discount);
```

注意，有时同一个运算符既可以用作一元运算符又可以用作二元运算符。例如，符号-和+既可以作为一元运算符，也可以作为二元运算符使用。

这些优先级规则与代数课上使用的规则类似。但是，除了一些非常标准的情况，即使所期望的运算顺序与优先级规则规定的顺序是一样的，最好也使用圆括号。对阅读程序代码的人来说，圆括号会使表达式更加清晰。在加法运算中包含乘法运算是一种可以将圆括号省略的标准情况。因此：

```
balance = balance + (interestRate * balance);
```

通常会写成

```
balance = balance + interestRate * balance;
```

这两种形式都是可以的，具有相同的含义。

书写算术表达式时，在运算符前后可以包含一些空格，也可以将其省略。同样，在圆括号周围也可以包含或不包含空格。

图2-5显示了一些例子，这些例子说明了如何在Java中书写算术表达式，并指出了一些通常可以省略的圆括号。

普通算术表达式	Java表达式（优先使用的形式）	等效的、全部使用了圆括号的Java表达式
$rate^2 + delta$	`rate * rate + delta`	`(rate * rate) + delta`
$2(salary + bonus)$	`2 * (salary + bonus)`	`2 * (salary + bonus)`
$\frac{1}{time + 3mass}$	`1/(time + 3*mass)`	`1/(time + (3*mass))`
$\frac{a-7}{t+9v}$	`(a-7)/(t + 9*v)`	`(a-7)/(t + (9*v))`

图2-5　Java中的算术表达式

案例分析：售货机的零钱问题

售货机中通常都由一些小型计算机来控制其操作。在这个案例研究中，要编写一个程序来处理这种计算机要执行的一项任务。输入和输出将通过键盘和屏幕来实现。要将这个程序集成到一台售货机的计算机中去，就要将这个程序的代码嵌入到一个更大的程序中，这个更大的程序会从键盘

之外的某个地方接收数据，并将结果发送到屏幕之外的某个地方。在这个案例研究中，用户输入1~99美分的零钱数。作为响应，程序要告知用户一种与用户输入的零钱数额相等的美分组合。

例如，如果用户输入“55”表示55美分，程序会告诉用户可以用两个25美分的硬币和一个5美分的硬币组成55美分。确定对话看起来应该如下例所示，这些是你在编写程序代码之前写下来看看程序的界面会是什么样子用的：

```
Enter a whole number from 1 to 99.
I will output a combination of coins
that equal that amount of change.
87
87 cents in coins:
3 quarters
1 dime
0 nickels and
2 pennies
```

程序需要使用变量来存储零钱数和每种类型的硬币数。因此，程序至少需要下列变量：

```
int amount, quarters, dimes, nickels, pennies;
```

这样就将一些常规问题处理好了，现在就可以去处理问题的核心了。这里需要一个算法来计算每种类型硬币的数量。

确定amount美分中硬币数量的算法

将数值总额读入变量amount中。
对变量quarters进行设置，使其等于amount中最多可以包含的25美分面值硬币的数量。
重新设置amount，使其等于给出了那么多25美分面值硬币之后剩下的零钱数。
对变量dimes进行设置，使其等于amount中最多可以包含的10美分面值硬币的数量。
重新设置amount，使其等于给出了那么多10美分面值硬币之后剩下的零钱数。
对变量nickels设置，使其等于amount中最多可以包含的5美分面值硬币的数量。
重新设置amount，使其等于给出了那么多5美分面值硬币之后剩下的零钱数。
pennies = amount;
输出原始的amount值和每种硬币的数量。

这个算法是用**伪代码**（pseudocode）表达的，伪代码是在将算法翻译成Java语言之前，用来表示算法的一种方式，它是由Java和自然语言以便于使用的方式组合起来的。

查看伪代码时，你会发现算法修改了amount的值。但是，你希望在算法结束时保留原始的总额，以便将其输出。因此，要用到另一个被称为originalAmount的变量来保存原始的总额。将算法改成下列形式。

确定amount美分中硬币数量的算法

将数值总额读入变量amount中。
originalAmount = amount;
对变量quarters进行设置，使其等于amount中最多可以包含的25美分面值硬币的数量。
重新设置amount，使其等于给出了那么多25美分面值硬币之后剩下的零钱数。
对变量dimes进行设置，使其等于amount中最多可以包含的10美分面值硬币的数量。
重新设置amount，使其等于给出了那么多10美分面值硬币之后剩下的零钱数。
对变量nickels进行设置，使其等于amount中最多可以包含的5美分面值硬币的数量。
重新设置amount，使其等于给出了那么多5美分面值硬币之后剩下的零钱数。
pennies = amount;

输出originalAmount和每种硬币的数量。

现在，需要生成Java代码来完成与伪代码相同的任务。大部分代码都是常规代码。伪代码中的第一行只是要求提示用户，然后从键盘读取输入。为第一行伪代码生成的Java代码如下所示。

```
System.out.println("Enter a whole number from 1 to 99.");
System.out.println("I will output a combination of coins");
System.out.println("that equals that amount of change.");

Scanner keyboard = new Scanner(System.in);
amount = keyboard.nextInt();
```

下一行伪代码设置了originalAmount的值，它已经是Java代码了，所以不需要进行任何转换。

到目前为止，程序的main部分如下所示。

```
public static void main(String[] args)
{
    int amount, originalAmount,
        quarters, dimes, nickels, pennies;
    System.out.println("Enter a whole number from 1 to 99.");
    System.out.println("I will output a combination of coins");
    System.out.println("that equals that amount of change.");

    Scanner keyboard = new Scanner(System.in);
    amount = keyboard.nextInt();

    originalAmount = amount;
```

接下来，要将下列伪代码转换成Java代码：

对变量quarters进行设置，使其等于amount中最多可以包含的25美分面值硬币的数量。

重新设置amount，使其等于给出了那么多25美分面值硬币之后剩下的零钱数。

55美分包含了2个25美分，因为55被25除得2，余5。所以可以用运算符/和%来做这种类型的除法。例如：

55/25等于2（55中最多可包含的25的数量）

55%25等于5（余数）

用amount取代55，并根据Java语法对其进行修改，可以生成下列代码：

```
quarters = amount/25;
amount = amount%25;
```

你会发现可以用类似的方式对10美分和5美分进行处理，这样，接下来就可以生成下列代码：

```
dimes = amount/10;
amount = amount%10;
nickels = amount/5;
amount = amount%5;
```

其余的程序代码都可以直接写出。这样，就可以生成图2-6中所示的程序作为最终程序了。

生成了程序之后，要用大量不同类型的数据对其进行测试。你决定用下列输入来测试程序：0美分、4美分、5美分、6美分、10美分、11美分、25美分、26美分、35美分、55美分、65美分及很多其他的情形。听起来好像有很多不同的输入，但你要尝试那些能使所有可能的硬币值都为0的情况，还要对那些靠近变化点的值进行测试，比如25美分和26美分，这些值中包含的硬币从全是25美分的硬币变成了25美分硬币与其他硬币的结合。所有测试都是成功的，但输出语句的语法却不完全正确。对26美分来说，得到的输出为：

```
26 cents in coins:
1 quarters
0 dimes
0 nickels and
1 pennies
```

```
import java.util.*;
public class ChangeMaker
{
    public static void main(String[] args)
    {
        int amount, originalAmount, quarters, dimes, nickels, pennies;

        System.out.println("Enter a whole number from 1 to 99.");
        System.out.println("I will output a combination of coins");
        System.out.println("that equals that amount of change.");

        Scanner keyboard = new Scanner(System.in);
        amount = keyboard.nextInt( );

        originalAmount = amount;
        quarters = amount/25;
        amount = amount%25;
        dimes = amount/10;
        amount = amount%10;
        nickels = amount/5;
        amount = amount%5;
        pennies = amount;

        System.out.println(originalAmount + " cents in coins can be given as:");
        System.out.println(quarters + " quarters");
        System.out.println(dimes + " dimes");
        System.out.println(nickels + " nickels and");
        System.out.println(pennies + " pennies");
    }
}
```

87中有3个25，还剩12。
87/25得3。
87%25得12。
87美分等于3个25美分，余12美分。

屏幕对话示例

```
Enter a whole number from 1 to 99.
I will output a combination of coins
that equals that amount of change.
87
87 cents in coins can be given as:
3 quarters
1 dimes
0 nickels and
2 pennies
```

图2-6 换零钱的程序

输出是正确的，但如果输出的是1 quarter而不是1 quarters，是1 penny而不是1 pennies，读起来感觉就好多了。在下一章中将对生成这种更好看的输出所需的技术进行介绍。现在，我们就此结束这个项目。输出是正确且可理解的。

自测题

12. 下列程序代码行会产生什么样的输出?

```
int quotient, remainder;
quotient = 7/3;
remainder = 7%3;
System.out.println("quotient = " + quotient);
System.out.println("remainder = " + remainder);
```

13. 下列程序代码行会产生什么样的输出?

```
double result;
result = (1/2) * 2;
System.out.println("(1/2) * 2 equals " + result);
```

14. 考虑图2-6程序中的下列语句:

```
System.out.println(originalAmount + " cents in coins can be given as:");
```

假设用下列语句取代前面那行代码:

```
System.out.println(amount + " cents in coins can be given as:");
```

会使图2-6中的对话示例发生什么变化?

15. 下列代码会产生什么样的输出?

```
int result = 11;
result /= 2;
System.out.println("result is " + result);
```

2.1.13 递增运算符和递减运算符

递增和递减运算符可以用来将变量的值加1或减1。它们是非常特殊的运算符，没有这些运算符，你（及Java）也可以很容易地编写程序。但使用这些运算符有时是很方便，因为很多程序员都使用这些运算符，所以它们也具有了一些文化含义。因此，要想“待在俱乐部里”，就应该了解如何使用这些运算符。即使自己不想使用它们，也需要熟悉这些运算符，这样，在别的程序员编写的代码中看到它们时，才能理解它们的含义。

递增运算符（increment operator）写成两个加号（++）。例如，下列代码会将变量count的值加1:

```
count++;
```

这是一条Java语句。如果在执行这条语句之前，变量count的值为5，那么执行了这条语句之后，它的值就是6了。可以对任意一种数字类型的变量使用递增运算符，但最常见的是将其用于整数类型的变量（比如int类型）。

递减运算符（decrement operator）与递增运算符类似，只是递减运算符是将变量的值减1而不是加1。递减运算符写成两个减号（--）。例如，下列代码会将变量count的值减1:

```
count--;
```

如果在执行这条语句之前变量的值为5，这条语句执行之后，其值就是4了。

注意:

```
count++;
```

等效于

```
count = count + 1;
```

而

```
count--;
```

等效于

```
count = count - 1;
```

递增和递减运算符确实是非常特殊的。为什么Java中会有这么特殊的运算符呢？Java是从C++继承而来的（而C++是从C继承的）。实际上，“C++编程语言”这个名字中的“++”就来自于递增运算符。为什么要向C和C++语言中添加这个运算符呢？因为加1或减1的操作是编程时很常见的一项工作。

2.1.14 更多关于递增运算符和递减运算符的内容

递增运算符和递减运算符可以用于表达式，但不推荐这样使用。将其用于表达式时，这些运算符都会修改应用它们的变量的值，并返回一个值。

在表达式中，可以将++或--放在变量前面或后面，但根据它是在变量之前还是之后会具有不同的含义。例如，考虑下列代码：

```
int n = 3;
int m = 4;
int result;
result = n * (++m);
```

执行之后，n的值为3，没有变化，m的值为5，result的值为15。因此，++m既修改了m的值，又将修改过的值返回到算术表达式中使用。

在前面的例子中，我们将递增运算符放在变量之前。如果将其放在变量m之后，发生的情况会略有不同。考虑下列代码：

```
int n = 3;
int m = 4;
int result;
result = n * (m++);
```

在这种情况下，执行了代码之后，n的值为3，m的值为5，和前面的情况一样，但是result的值为12，而不是15。这是怎么回事呢？

n*(++m)和n*(m++)这两个表达式都将m的值加了1，但第一个表达式是在做乘法之前增加了m的值，而第二个表达式则是在做乘法之后增加了m的值。++m和m++对m最终值的影响是相同的，但将其作为算术表达式的一部分使用时，它们传递给表达式的值是不同的。如果++在m之前，m的值会在将其值用于表达式之前增加；如果++在m之后，m的值就会在将其值用于表达式之后增加。

将--运算符用于算术表达式时，会以类似的方式工作。--m和m--对m最终值的影响是相同的，但将它们作为算术表达式的一部分使用时，传递给表达式的值是不同的。如果--在m之前，m的值会在将其值用于表达式之前减少。如果--在m之后，m的值就会在将其值用于表达式之后减少。

递增运算符和递减运算符只能用于变量，不能用于常量或更复杂的算术表达式。

自测题

16. 下列程序代码行会产生什么样的输出？

```
int n=2;
n++;
System.out.println("n == " + n);
n--;
System.out.println("n == " + n);
```

2.2 String类

空话，空话，只有空话，没有一点真心。

——威廉·莎士比亚，《特洛埃勒斯与克蕾雪达》

对"Enter the amount:"这样的字符串的处理与对基本类型值的处理略有不同。Java中没有用于字符串的基本类型，但是有一个名为String的类可以用来存储和处理字符串。在这一节中，我们将向你介绍String类。

2.2.1 字符串常量和变量

前面已经使用过String类型的常量了。图2-6程序中出现了下列语句：

```
System.out.println("Enter a whole number from 1 to 99.");
```

其中的引用字符串

```
"Enter a whole number from 1 to 99."
```

就是一个字符串常量。

String类型的值就是这样一个引用字符串，即String类型的值是被当作一个单独项处理的字符序列。可以用String类型的变量来命名这样一个字符串的值。

下列语句将greeting声明为一个String变量名：

```
String greeting;
```

下列语句将greeting的值设置为String值"Hello!"：

```
greeting = "Hello!";
```

通常会将这两条语句合并为一条语句，如下所示：

```
String greeting = "Hello!";
```

一旦为greeting这样的String变量赋值，就可以按下列方法将其写到屏幕上：

```
System.out.println(greeting);
```

如果已经按照我们刚才描述的方法设置了greeting的值，这条语句就会将

```
Hello!
```

写到屏幕上。

2.2.2 字符串的拼接

可以用+运算符将两个字符串连接起来。将两个字符拼接（“粘贴”）到一起以获得一个更长的字符串，这种方法被称为**拼接**（concatenation）。因此，当符号+用于字符串时，有时也称为**拼接运算符**（concatenation operator）。例如：

```
String greeting = "Hello";
String sentence;
sentence = greeting + "my friend.";
```

```
System.out.println(sentence);
```

这些语句会将变量sentence设置为"Hellomy friend."，并且会将下列内容写到屏幕上：

```
Hellomy friend.
```

注意，通过+运算符将两个字符串拼接起来时，没有添加空格。如果想将sentence设置为"Hello my friend."，可以将赋值语句改成：

```
sentence = greeting + " my friend.";
```

注意，单词“my”前面的空格。

可以用+运算符拼接任意数量的String对象。甚至可以通过+运算符将一个String变量和任何其他类型的对象拼接起来。结果总是一个String对象。用+运算符将任意的对象连接到一个字符串时，Java都会找出某种方法将其表示为一个字符串。对数字这种简单的项，它会使用显而易见的方法。例如：

```
String solution = "The answer is " + 42;
```

会将String变量solution设置为"The answer is 42"。这是非常自然的，好像没发生什么特别的事情。但它确实需要一种类型到另一种类型的转换。常量42是一个数字，而"42"是一个字符串，由字符'4'后面跟着字符'2'组成。Java将数字常量42转换成字符串常量"42"，然后将两个字符串"The answer is "和"42"拼接起来，以获取更长的字符串"The answer is 42"。

快速参考：为字符串使用+号

可以用+号将两个字符连接起来以实现两者的拼接。

举例：

```
String name = "Chiana";
String greeting = "Hi " + name;
System.out.println(greeting);
```

这个例子将greeting设置为字符串"Hi Chiana"，然后将下列内容输出到屏幕：

```
Hi Chiana
```

注意，要在"Hi"后面加一个空格。

2.2.3 类

类对Java来说是很重要的，你很快就会更多地定义和使用自己的类了。但是，这里对String类的讨论为我们提供了一个介绍在类中使用的表示法和术语的机会。**类**是一种以对象为值的类型。**对象**是一些可以存储数据并执行动作的实体。例如，String类的对象就存储了由字符串组成的数据，比如"Hello"。对象可以执行的动作称为**方法**（method）。String类的大部分方法都会返回（或生成）某些值。例如，方法length()会返回一个Sting对象中的字符数。因此"Hello".length()会返回整数5，并可以通过下列形式将其存储在一个int变量中：

```
int n = "Hello".length();
```

正如例子"Hello".length()指出的那样，可以写下对象的名字，后面跟一个点，后面再跟方法名，并以圆括号结尾，以此来调用方法执行动作。在你调用方法执行动作时，就称**调用了方法**，点前面的对象称为**调用对象**（calling object）。

尽管可以像"Hello".length()这样用常量对象来调用方法，但更常见的是按照如下所示

的方法，用命名了一个对象的变量作为调用对象：

```
String greeting = "Hello";
int n = greeting.length();
```

传递给方法调用的信息是在圆括号中给出的。在某些情况下，比如在方法length中，不需要什么信息（除了调用对象中的数据之外），圆括号中是空的。在其他情况下，就像你马上会看到的那样，必须在圆括号中提供一些信息。圆括号中的信息被称为**实参**（argument）。

类中所有的对象都具有相同的方法，但每个对象都可以有不同的数据。例如，两个String对象"Hello"和"Good-bye"就有不同的数据，即不同的字符串。但是，它们具有相同的方法。因此，由于我们知道String对象"Hello"拥有方法length()，我们就知道String对象"Good-Bye"一定也拥有方法length()。

快速参考：对象、方法和类

对象是一种程序结构，它拥有与之相关的数据（即信息），并可以执行某些动作。对象执行的动作称为**方法**。**类**是一类对象。同一个类中的不同对象可以有不同的数据，但同一个类中的所有对象都拥有相同类型的数据和相同的方法。当对象执行一个方法的动作时，就称对象**调用**了方法。方法调用的语法是：在对象名，后面跟一个点，再跟上方法名，然后以一个可能为空的圆括号对结束。圆括号中包含了一些调用方法时要用的信息。圆括号中的这些信息被称为方法的实参。

现在，你已经看到了Java中的两种类型：基本类型和类类型。它们之间的主要区别就是类类型有方法，而基本类型没有方法。一个较小的区别就是所有的基本类型都仅用小写字母拼写；而按照惯例，类类型的第一个字母都是大写的，就像String一样。稍后，你还会看到类类型和基本类型之间更多的区别。

2.2.4 字符串方法

String变量不只是一个int类型变量那样的简单变量。String变量是一个类类型的变量，它可以命名一个对象，而对象是有方法和值的。可以用这些String方法来操纵字符串的值。图2-7中描述了几个String方法。和所有的方法一样，可以在对象名之后写一个点和方法名来调用String方法。在本节中，对象名是 String类型的变量。方法的所有实参都在圆括号中给出。下面来看一些例子。

正如我们指出的那样，可以用方法length查明一个字符串中字符的数量。例如，假设我们按如下所示声明了一些String变量：

```
String command = "Sit Fido!";
String answer = "bow-wow";
```

那么，command.length()会返回9，而answer.length()会返回7。注意，计算字符串长度时，空格、特殊符号及重复的字符都计算在内了。

可以在任何能够使用int类型值的地方使用length方法的调用。例如，下列所有语句都是合法的Java语句。

```
int count = command.length();
System.out.println("Length is " + command.length());
count = command.length() + 3;
```

方　法	描　述	举　例
length()	返回String对象的长度	String greeting = "Hello!"; greeting.length()返回6。
equals(*Other_String*)	如果调用对象字符串与*Other_String*相同，就返回true，否则返回false	String greeting = keyboard.next(); if (greeting.equals("Hi")) System.out.println("Informal Greeting.");
equalsIgnoreCase (*Other_String*)	在不区分字母的大写和小写的情况下，如果调用对象字符串与*Other_String*相同，返回true，否则返回false	如果程序中包含 String s1 = "mary!"; 那么，在这条赋值语句之后，s1.equalsIgnoreCase("Mary!")会返回true
toLowerCase()	返回与调用对象字符串具有相同字符的字符串，但会将所有字符都转换成小写字符	String greeting = "Hi Mary!"; greeting.toLowerCase()会返回"hi mary!"
toUpperCase()	返回与调用对象字符串具有相同字符的字符串，但会将所有字符都转换成大写字符	String greeting = "Hi Mary!"; greeting.toUpperCase()会返回"HI MARY!"
trim()	返回与调用对象字符串具有相同字符的字符串，但删除了所有开始和结尾处的空格	String pause = "Hmm"; pause.trim()会返回"Hmm"
charAt(*Position*)	返回调用对象字符串中位于*Position*的字符。位置是按0、1、2计数的	String greeting = "Hello!"; greeting.charAt(0) 会返回'H'. greeting.charAt(1) 会返回'e'
substring(*Start*)	返回调用对象字符串中从位置*Start*开始，到调用对象结尾为止的子字符串。位置是按0、1、2计数的	String sample = "AbcedfG"; sample.substring(2) 会返回"cdefG".
substring(*Start, End*)	返回调用对象字符串中从位置*Start*开始，到调用对象的位置*End*为止，但不包括位置*End*的子字符串。位置是按0、1、2计数的	String sample = "AbcedfG"; sample.substring(2,5) 会返回"cde"
indexOf(*A_String*)	返回在调用对象字符串中第一次出现字符串*A_String*的位置。位置是按0、1、2计数的。如果没找到*A_String*，返回−1	String greeting = "Hi Mary!"; greeting.indexOf("Mary") 会返回3 greeting.indexOf("Sally") 会返回-1
indexOf(*A_String, Start*)	返回从位置*Start*起，或*Start*之后，调用对象字符串中第一次出现字符串*A_String*的位置。位置是按0、1、2计数的。如果没找到*A_String*，返回−1	String name = "Mary, Mary quite contrary"; name.indexOf("Mary",1)会返回6。如果使用任意到6为止、包括6在内的数来取代1，都会返回相同的结果 name.indexOf("Mary",0) 会返回0 name.indexOf("Mary",8) 会返回-1
lastIndexOf(*A_String*)	返回调用对象字符串中最后一次出现字符串*A_String*的位置。位置是按0、1、2计数的。如果没找到*A_String*，返回−1	String name = "Mary, Mary, Mary quite so"; name.lastIndexOf("Mary")会返回12
compareTo(*A_String*)	比较调用对象字符串与*A_String*，看看按照词典顺序，哪个在前面。当两个字符串都是大写或都是小写时，词典顺序和字母顺序是一样的。如果调用字符串在前，compareTo返回一个负数。如果两个字符串相同，就返回0。如果*A_String*在前，返回一个正数	String entry = "adventure"; entry.compareTo("zoo") 会返回一个负数 entry.compareTo("adventure") 会返回零 entry.compareTo("above") 会返回一个正数

图2-7 String类中的方法

String类中很多方法都依赖于对字符串中**位置**（position）的计数。位置的计数是从0开始，而不是从1开始。因此，在字符串"Hi Mom"中，'H'位于位置0，'i'位于位置1，空白字符位于位置2，依次类推。在计算机用语中，通常将位置称作**索引**（index）。因此，更常见的用法是说'H'位于索引0，'i'位于索引1，空白字符位于索引2，等等。图2-8说明了在字符串中是如何对索引位置进行编号的。

字符串"Java is fun."中12个字符的索引为0~11。在每个字符的上面显示的是其索引。

0	1	2	3	4	5	6	7	8	9	10	11
J	a	v	a		i	s		f	u	n	.

注意，在字符串中空格和句点都是作为字符计数的。

图2-8　字符串索引

方法indexOf会返回作为其实参给出的子字符串的索引。如果子字符串多次出现，indexOf会返回其子字符串实参第一次出现时的索引。例如：

```
String phrase = "Time flies like an arrow.";
```

经过这样的声明之后，由于"flies"中的'f'位于索引5，所以，调用phrase.indexOf("flies")会返回5（记住，第一个索引是0，而不是1。）

2.2.5　字符串处理

很多Java语言的参考书都说String类型的对象是无法修改的。在某种意义上，这种说法是对的，但这是一种误导性的说法。注意，图2-7中任何方法都未修改对象String的值。String方法比图2-7中显示的要多，但没有一种方法能让你写出具有"将调用对象字符串中的第5个字符改成'z'"这种含义的语句。这不是一个意外。这是为了使String类的实现更有效。也就是说，这是为了使方法执行得更快、使用更少的计算机存储器，而有意设计的。但是，另一种字符串类中拥有修改字符串对象的方法。这个类被称为StringBuffer，但我们现在并不需要这个类，所以我们不在这里对其进行讨论。

尽管没有方法可以修改"Hello"这样的String对象的值，但仍然可以编写修改了String变量值的程序。要进行修改，只要像下面的例子一样，使用一条赋值语句就可以了：

```
String name = "D'Aargo";
name = "Ka " + name;
```

第二行的赋值语句将变量name的值由"D'Aargo"改成了"Ka D'Aargo"。图2-9显示了一个示例程序，这个程序说明了如何进行一些简单的字符串处理，以改变一个String变量的值。2.2.6节对部分程序进行了解释。

```
public class StringDemo
{
    public static void main(String[] args)
    {
        String sentence = "Text processing is hard!";
        int position;
        position = sentence.indexOf("hard");
        System.out.println(sentence);
        System.out.println("012345678901234567890123");
        System.out.println("The word \"hard\" starts at index "
                                + position);
        sentence = sentence.substring(0, position) + "easy!";
        System.out.println("The changed string is:");
        System.out.println(sentence);
    }
}
```

屏幕对话示例

```
Text processing is hard!
012345678901234567890123
The word "hard" starts at index 19
The changed string is:
Text processing is easy!
```

\"的含义将在2.2.6节中进行讨论。

图2-9 使用String类

2.2.6 转义符

假设你想输出一个包含引号的字符串。比如，想将下列内容输出到屏幕上：

```
The word "Java" names a language, not just a drink!
```

下列代码是无效的：

```
System.out.println("The word "Java" names a language, not just a drink!");
```

这条语句会产生一条编译器错误消息。问题在于编译器会将"The word "当成一个完全合法的引用字符串。然后，编译器会看到Java"（尽管编译器可能会猜它是个丢失了一个引号的引用字符串，或者会猜你漏掉了一个+号）。这在Java语言中是不合法的。除非你告诉编译器你想将符号'"'作为引用字符串的一部分包含进来，否则编译器无法知道你想干什么。你可以在那个字符前面放一个反斜杠（\），就可以通知编译器你想将引号包含到字符串中，如下所示。

```
System.out.println(
    "The word \"Java\" names a language, not just a drink!");
```

图2-10列出了其他一些用反斜杠指示的特殊字符。由于它们脱离了字符的常用含义（比如双引号的常用含义），因此这些字符常被称为**转义序列**（escape sequence）或**转义符**（escape character）。

需要注意的很重要的一点是，尽管会将每个转义符都写成两个符号，但实际上它是一个

单字符。因此字符串"Say \"Hi\"!"包含了9个而不是11个字符（S、a、y、空白字符、\"、H、i、\"和!）。

\"	双引号
\'	单引号
\\	反斜杠
\n	新行。转到下一行的起始位置
\r	回车符。转到当前行的起始位置
\t	制表符。添加空格，直到下一个制表位为止

图2-10　转义符

要在引用字符串中包含一个反斜杠会有点儿棘手。比如，字符串"abc\def"很可能会生成错误信息“非法的转义符”。要在字符串中包含一个反斜杠，就要使用两个反斜杠。如果将字符串"abc\\def"输出到屏幕上去，会生成

```
abc\def
```

转义符\n意味着字符串会在\n处开始一个新行。例如，语句

```
System.out.println("The motto is \nGo for it!");
```

会将下列内容写到屏幕上：

```
The motto is
Go for it!
```

像"How's this?"这样在引用字符串中包含单引号是完全合法的，所以看起来好像没必要使用转义符\'。但是，如果要为单引号字符指定一个常量，就需要使用转义符，例如：

```
char singleQuote = '\'';
```

2.2.7 Unicode字符集

大多数其他编程语言都使用附录C中给出的ASCII字符集。ASCII字符集仅仅是通常用于英文键盘的所有字符的列表，以及分配给这些字符的标准编号。Java使用的则是Unicode字符集。Unicode字符集包含了整个ASCII字符集，并添加了很多在其他语言中使用的字符，这些语言的字母表通常与英语不同。这样，如果使用英语键盘，就不会有什么大问题。通常，编程时可以认为Java使用的是ASCII字符集，因为ASCII字符集是Unicode字符集的子集。因此，可以将附录C当作ASCII字符集的列表，也可以将它当作Unicode字符集子集的列表。Unicode字符集的优点是很容易用它来处理英语之外的其他语言。缺点是，与只使用ASCII字符集相比，有时需要更多的计算机存储器空间来存储每个字符。

自测题

17. 下列代码会产生什么样的输出?

```
String.greeting = "How do you do";
System.out.println(greeting + "Seven of Nine.");
```

18. 下列代码会产生什么样的输出?

```
String test = "abcdefg";
System.out.println(test.length());
System.out.println(test.charAt(1));
```

19. 下列代码会产生什么样的输出?

```
String test = "abcdefg";
System.out.println(test.substring(3));
```

20. 下列代码会产生什么样的输出?

```
System.out.println("abc\ndef");
```

21. 下列代码会产生什么样的输出?

```
System.out.println("abc\\ndef");
```

22. 下列代码会产生什么样的输出?

```
String test = "Hello John";
test = test.toUpperCase();
System.out.println(test);
```

23. 下列代码会产生什么样的输出?

```
String s1 = "Hello John";
String s2 = "hello john";
if (s1.equals(s2))
   System.out.println("Equal");
System.out.println("End");
```

在第3章之前都没有对if进行解释，但根据if在日常英语中的含义完全可以理解这种简单的情况。

24. 下列代码会产生什么样的输出?

```
String s1 = "Hello John";
String s2 = "hello john";
s1 = s1.toUpperCase();
s2 = s2.toUpperCase();
if(s1.equals(s2))
   System.out.println("Equal");
System.out.println("End");
```

2.3 键盘与屏幕I/O

输入的是垃圾，输出的也将是垃圾。

——**程序员谚语**

程序数据的输入和输出通常被称为I/O。Java程序有很多不同的方式来执行I/O。在本节中，我们提供了一些非常简单的方法，对从键盘输入的文本输入和发送到屏幕的文本输出进行处理。后续各章将会介绍一些更复杂的I/O处理方法。

2.3.1 屏幕输出

我们从本书的开始就在使用简单的输出语句了。本节只是对已经在做的工作进行了总结和解释。在图2-6中，使用了如下所示的语句将输出发送到显示屏幕上：

```
System.out.println("Enter a whole number from 1 to 99.");
          ...
System.out.println(quarters + " quarters");
```

System.out是一个对象，这个对象是Java语言的一部分。拼写一个带点的对象名可能会让人觉得很奇怪，不过现在你还不需要关心这个问题。

对象System.out将println作为它的一个方法。因此，前面的输出语句是对System.out对象的println方法的调用。当然，并不需要为了使用这些输出语句而去了解这些细节。可

以只将System.out.println当作一条特殊拼写的语句。但你最好习惯这种点表示法以及方法和对象的表示法。

要使用这种形式的输出语句，只要在表达式System.out.println后面跟上你想输出的内容，并用圆括号把它括起来，最后再加一个分号就行了。可以输出双引号中的文本字符串，比如"Enter a whole number from 1 to 99."或" quarters"；可以输出quareters这样的变量；可以输出5或7.3这样的数字；还可以输出几乎所有其他对象和值。如果你想输出的内容多于一项，只要在你想输出的内容之间放置一个加号就行了。例如：

```
System.out.println("Lucky number = " + 13  + "Secret number = " + number);
```

如果number的值为7，输出将如下所示。

```
Lucky number = 13Secret number = 7
```

注意，这里没有添加任何空格。如果希望在前面的输出中，在13和单词Secret之间有一个空格，就应该在字符串

```
"Secret number = "
```

的开头加一个空格。这样它就变成了

```
" Secret number = "
```

注意，你使用的是双引号，而不是单引号，而且左边和右边的引号都是相同的符号。最后要注意，如果一条语句太长，可以将它放在两行上。但是，考虑到可读性，应该将第二行缩进，并且应该在+号之前或之后断行。不能在一个引用字符串或变量名的中间断行。

也可以用println方法来输出一个String变量的值，如下列语句所示。

```
String greeting = "Hello Programmers!";
System.out.println(greeting);
```

这些语句会将下列内容写到屏幕上：

```
Hello Programmers!
```

每个println调用都会结束一个输出行。例如：

```
System.out.println("One, two, buckle my shoe.");
System.out.println("Three, four, shut the door.");
```

这两条语句会使下列输出出现在屏幕上：

```
One, two, buckle my shoe.
Three, four, shut the door.
```

如果你希望两条或多条输出语句将它们所有的输出都放在同一行，就用print取代println。例如：

```
System.out.print("One, two,");
System.out.print(" buckle my shoe.");
System.out.println(" Three, four,");
System.out.println(" shut the door.");
```

会产生下列输出：

```
One, two, buckle my shoe. Three, four,
 shut the door.
```

注意，只有使用println（而不是print），才会开始一个新行。还要注意，新行是在输出了println中指定的项之后才开始的。这是print和println之间唯一的区别。

这就是用这种类型的输出来编写程序时所需了解的全部内容了，但我们还可以对输出语句中发生的情况进行更多的解释。考虑下列语句：

```
System.out.println("The answer is " + 42);
```

圆括号中的表达式"The answer is " + 42看起来很熟悉。

在2.2.4节中，我们说过可以用+运算符将一个字符串（如"The answer is "）和另一个项（如数字常量42）连接起来。这些System.out.println语句中的+运算符与执行字符串连接的+运算符是一样的。在System.out.println语句中，Java将数字常量42转换成字符串"42"，然后用+运算符得到了字符串"The answer is 42"。然后System.out.println语句输出字符串"The answer is 42"。println方法输出的总是字符串。从技术上来说，它决不会输出数字，即使看起来它好像输出了数字。

快速参考：println输出

可以用System.out.println来输出一行内容。输出的项可以是加了引号的字符串、变量、数字这样的常量，或者几乎所有能在Java中定义的对象。

语法：

System.out.println(*Output_1* + *Output_2* + ··· + *Output_Last*);

举例：

```
System.out.println("Hello out there!");
System.out.println("Area = " + theArea + " square inches");
```

快速参考：println与print

System.out.println和System.out.print基本上是相同的方法。唯一的区别就是使用方法println时，下一个输出会出现在一个新行上，而使用方法print时，下一输出则会被放在同一行上。

```
System.out.print("one ");
System.out.print("two ");
System.out.println("three ");
System.out.print("four ");
```

会产生下列输出：

```
one two three
four
```

无论最后一行使用的是print还是println，输出看起来都是一样的。

2.3.2 键盘输入

从版本1.5开始，Java为键盘输入的处理提供了一些合理的功能。这些功能是通过java.util包中的Scanner类来实现的；util是*utility*（实用程序）的缩写，但在Java代码中，通常都使用简写util。包就是一个类库。要在程序中使用Scanner（及java.util包中的所有其他类），就必须在包含了你所编写程序的文件的起始位置附近插入下列行：

```
import java.util.*;
```

键盘输入是由Scanner类的对象实现的。可以按照下列方式创建一个Scanner类的对象：

Scanner *Scanner_Object_Name* = new Scanner(System.in);

其中*Scanner_Object_Name*是任意的（非关键字）Java标识符。例如，在图2-6中，用标识符keyboard作为*Scanner_Object_Name*：

```
Scanner keyboard = new Scanner(System.in);
```

由于Scanner类是用于键盘输入的，所以通常会为Scanner对象使用标识符keyboard，但是，你可以使用其他名字。如果你希望你的Scanner类对象被命名为scannerObject，就应该使用下列语句：

```
Scanner scannerObject = new Scanner(System.in);
```

在这行代码之后，就可以通过对象scannerObject使用Scanner类的方法来读取用户在键盘上输入的数据了。例如，方法nextInt会读入一个在键盘上输入的int值。方法调用

```
scannerObject.nextInt()
```

会从键盘读入一个int值，并返回int值。可以按下列方式将从键盘读入的值赋给一个int类型的变量n1：

```
n1 = scannerObject.nextInt();
```

图2-11对此进行了说明。注意，从键盘输入的任意两个数字都必须用空白字符隔开，如一个或多个空格，或者一个或多个换行符，或者空格与换行符的组合。

```
import java.util.*;            ← 包含Scanner类的包（库）名。
public class ScannerDemo
{
    public static void main(String[] args)
    {
        int n1, n2;                            ← 创建对象，使得程序可以接受键盘输入。
        Scanner scannerObject = new Scanner(System.in);

        System.out.println("Enter two whole numbers");
        System.out.println("seperated by one or more spaces:");

        n1 = scannerObject.nextInt( );   ← 从键盘读入一个int类型值。
        n2 = scannerObject.nextInt( );
        System.out.println("You entered " + n1 + " and " + n2);

        System.out.println("Next enter two numbers.");
        System.out.println("A decimal point is OK.");

        double d1, d2;
        d1 = scannerObject.nextDouble( );   ← 从键盘读入一个double类型值。
        d2 = scannerObject.nextDouble( );
        System.out.println("You entered " + d1 + " and " + d2);
        System.out.println("Next enter two words:");

        String s1, s2;
        s1 = scannerObject.next( );   ← 从键盘读入一个单词。
        s2 = scannerObject.next( );
        System.out.println("You entered \"" +
                           s1 + "\" and \"" + s2 + "\"");

        s1 = scannerObject.nextLine( ); //To get rid of '\n'
                                         ↑ 在下面：易犯错误“方法nextLine带来的问题”中对这行代码进行了解释。
        System.out.println("Next enter a line of text:");
        s1 = scannerObject.nextLine( );   ← 读入一整行。
        System.out.println("You entered: \"" + s1 + "\"");
    }
}
```

图2-11　键盘输入示例

屏幕对话示例

```
Enter two whole numbers
separated by one or more spaces:
    42    43
You entered 42 and 43
Next enter two numbers.
A decimal point is OK.
    9.99  21
You entered 9.99 and 21.0
Next enter two words:
plastic spoons
You entered "plastic" and "spoons"
Next enter a line of text:
May the hair on your toes grow long and curly.
You entered "May the hair on your toes grow long and curly."
```

图2-11 （续）

如果要读入除了int之外的其他一些类型的数字，该怎么办呢？方法nextDouble的工作方式与nextInt完全相同，只是它读入的是double类型的值。图2-11对此进行了说明。图2-12给出了用来读入其他数字类型值的类似方法。

可以用Scanner类从文件获取输入，也可以用它从键盘获取输入。但在这里假定只用它从键盘获取输入。

要想使用Scanner类，就要在带有键盘输入代码的文件开头添加下列代码：

import java.util.*;

还要在第一条键盘输入语句之前加入下列代码：

Scanner *Scannner_Object_Name* = new Scanner(System.in);

Scannner_Object_Name.next()

返回String值，它包含了从下一个键盘字符开始，到第一个定界符为止的字符，但不包括这个定界符。默认的定界符为空白字符

Scannner_Object_Name.nextLine()

返回当前键盘输入行上的其余字符，并将这些字符作为String类型的值返回。要注意的是，行终止符'\n'被读入并丢弃，没有包含在返回的字符串中。

Scannner_Object_Name.nextInt()

返回键盘上输入的下一个int类型的值

Scannner_Object_Name.nextLong()

返回键盘上输入的下一个long类型的值

Scannner_Object_Name.nextByte()

返回键盘上输入的下一个byte类型的值

Scannner_Object_Name.nextShort()

返回键盘上输入的下一个short类型的值

Scannner_Object_Name.nextDouble()

返回键盘上输入的下一个double类型的值

Scannner_Object_Name.nextFloat()

返回键盘上输入的下一个float类型的值

Scannner_Object_Name.nextBoolean()

返回键盘上输入的下一个boolean类型的值。true或false值是作为字符串"true"和"false"输入的。在拼写"true"和"false"时可以使用任何大小写字母的组合

图2-12 Scanner类中的方法

Scannner_Object_Name.`useDelimiter(`*Delimiter_Word*`);`

使字符串*Delimiter_Word*成为可以用来分隔输入的唯一定界符。只有与之完全相同的单词才是定界符。特别是，空格、回车符及其他空白字符都不再是定界符了（除非它们是*Delimiter_Word*的一部分）。

这是useDelimiter方法的一种简单应用。有很多方式可以将定界符设置为各种不同字符和单词的组合，但在本书中不对其进行讨论。

图2-12　（续）

如图2-11中的以下代码所示，方法next会读入一个单词：

```
String s1, s2;
s1 = scannerObject.next();
s2 = scannerObject.next();
```

如果输入行为

```
plastic spoons
```

那么，这段代码会将字符串"plastic"赋给s1，将字符串"spoons"赋给s2。

对方法next来说，**单词**（word）就是任意一个由空格这样的空白字符或行的开始或结束来定界的、无空白字符字符串。

要读入一整行，就应该使用方法nextLine。例如：

```
String sentence;
sentence = scannerObject.nextLine();
```

会读入一个输入行，并将读入的字符串放到变量sentence中。

输入行的结束是由转义符'\n'标识的。'\n'字符就是在键盘上按下Enter键时输入的内容。在屏幕上是通过一行的结束和下一行的开始来标识的。nextLine读入一行文本时，会读入'\n'字符，但'\n'并不会成为返回的字符串值的一部分。因此，在前面的代码中，由变量sentence命名的字符串不是以'\n'字符结尾的。

快速参考：使用Scanner类的键盘输入

可以用Scanner类的对象读取键盘输入。要使用Scanner类，就要在包含所编写程序（或其他实现键盘输入的代码）的文件的起始处包含下列代码：

```
import java.util.*;
```

而且，代码还必须在读取任何键盘输入之前包含下列语句：

Scanner *Scannner_Object_Name* = new Scanner(System.in);

其中，*Scannner_Object_Name*可以是任意的（非关键字）Java标识符。例如：

```
Scanner scannerObject = new Scanner(System.in);
```

方法nextInt、nextDouble和next分别会读入一个int类型的值、一个double类型的值和一个单词。方法nextLine会读入包含终止符'\n'在内的当前输入行上其余的字符；但在返回的字符串值中是不包含'\n'的。图2-12中给出了其他输入方法。

语法：

Int_Variable = *Scannner_Object_Name*.nextInt();
Double_Variable = *Scannner_Object_Name*.nextDouble();
String_Variable = *Scannner_Object_Name*.next();
String_Variable = *Scannner_Object_Name*.nextLine();

举例：

```
int count;
count = scannerObject.nextInt();
```

```
double distance;
distance = scannerObject.nextDouble();
String word;
word = scannerObject.next();
String wholeLine;
wholeLine = scannerObject.nextLine();
```

记住：输入提示

程序需要用户输入一些数据时，要像下面的例子一样提示用户：

```
System.out.println("Enter a whole number:");
```

▲ 易犯错误：方法nextLine带来的问题

无论上一次键盘读入是在哪个地方停止的，Scanner类的方法nextLine都会从这个地方开始读取文本行中的剩余内容（remainder）。例如，假设你按下列方法创建了一个Scanner类的对象：

```
Scanner scannerObject = new Scanner(System.in);
```

并假设继续编写了下列代码：

```
int n;
n = scannerObject.nextInt();
String s1, s2;
s1 = scannerObject.nextLine();
s2 = scannerObject.nextLine();
```

最后，假设来自键盘的相应输入为

```
42 is the answer
and don't you
forget it.
```

这样，就会将变量n的值设置为42，将变量s1的值设置为“is the answer”，将变量s2的值设置为“and don't you”。

到目前为止，看起来好像不存在任何潜在的问题，但是，假设输入为

```
42
and don't you
forget it.
```

在这种情况下，你可能期望变量n的值被设置为42，变量s1的值被设置为"and don't you"，变量s2的值被设置为"forget it."。但是，事实却并非如此。

实际发生的情况是：变量n的值被设置为42，变量s1的值被设置为空字符串，变量s2的值被设置为"and don't you"。方法nextInt读入了42，但并没有读入行结束字符'\n'。因此，第一条nextLine调用读入的是42所在行的剩余部分。除了'\n'之外，这一行就没有任何内容了，因此，nextLine会返回一个空字符串。方法nextLine读入并丢弃了行结束字符'\n'。因此，下一条nextLine调用会从下一行开始，并读入"and don't you"。

结合不同的方法从键盘读取输入时，有时不得不包含一条额外的nextLine调用，以去除行的结束符（以去除'\n'）。图2-11对这个问题进行了说明。 △

记住：空字符串

一个字符串可以包含任意数量的字符。例如，"Hello"中包含5个字符。包含0个字符的字符串被

称为**空字符串**（empty string）。书写空字符串时，要像""这样在一对双引号之间不包含任何内容。遇到空字符串的机会并不少。如果代码执行nextLine方法读入一行文本时，用户在这一行上除了输入Enter（Return）键之外没有输入任何内容，那么，方法nextLine读入并返回的就是一个空字符串。

■ Java提示：其他输入定界符（选读）

为键盘输入使用Scanner类时，几乎可以将分隔键盘输入的定界符改成任何字符和字符串的组合，但实现的细节有些复杂。在本书中，我们只描述一种简单的定界符修改方式，即如何将定界符从空白字符改成一个指定的定界符字符串。

例如，假设你按下列方式创建了一个Scanner对象：

```
Scanner keyboard2 = new Scanner(System.in);
```

可以按照下列方式将对象keyboard2的定界符修改成"##"：

```
keyboard2.useDelimiter("##");
```

这次useDelimiter方法调用之后，"##"就会成为输入对象keyboard2唯一的输入定界符。注意，空白字符不再是用keyboard2实现的键盘输入的定界符了。因此，如果给出的键盘输入为：

```
funny wor##rd##
```

那么，下列代码将读入两个字符串"funny wor"和"rd"：

```
String s1, s2;
System.out.println("Enter a line of text with two words:");
s1 = keyboard1.next();
s2 = keyboard1.next();
```

图2-13对此进行了说明。注意，一旦对keyboard2进行了这种修改，空白字符甚至换行符都不会被当成输入定界符了。还要注意，在同一个程序中可以有两个使用不同定界符的Scanner类对象。

```
import java.util.*;

public class DelimitersDemo
{
    public static void main(String[] args)
    {
        Scanner keyboard1 = new Scanner(System.in);
        Scanner keyboard2 = new Scanner(System.in);

        keyboard2.useDelimiter("##");
        //The delimiters for keyboard1 are the whitespace characters.
        //The only delimiter for keyboard2 is ##.

        String s1, s2;

        System.out.println("Enter a line of text with two words:");
        s1 = keyboard1.next();
        s2 = keyboard1.next();
        System.out.println("the two words are \"" + s1
                              + "\" and \"" + s2 + "\"");

        System.out.println("Enter a line of text with two words");
        System.out.println("delimited by ##:");
        s1 = keyboard2.next();
```

keyboard1和keyboard2具有不同的定界符。

图2-13　修改定界符（选读）

```
        s2 = keyboard2.next();
        System.out.println("the two words are \"" + s1
                          + "\" and \"" + s2 + "\"");
    }
}
```

屏幕对话示例

```
Enter a line of text with two words:
funny wo##rd##
The two words are "funny" and  "wor##rd##"
Enter a line of text with two words
delimited by ##:
funny wor##rd##
The two words are "funny wo" and "rd"
```

图2-13 （续）

自测题

25. 编写能向屏幕写出下列内容的Java语句。

```
Once upon a time,
there were three little programmers.
```

26. `System.out.println`和`System.out.print`之间有什么区别？
27. 编写一个完整的Java程序，使其读入一个包含了两个（由一个或多个空格分隔的）`int`类型值的键盘输入行，并将这两个数字输出。
28. 编写一个完整的Java程序，使其读入一个恰好包含了3个（由任意类型及数量的空白字符分隔的）单词的文本行，并将经过间隔校正的行输出；即输出的行中第一个单词前面没有空格，并且每对相邻的单词之间恰好有两个空格。
29. 下列代码会产生什么样的输出？

```
String s = "Hello" + "" + "Joe";
System.out.println(s);
```

2.4 文档与样式

矮胖子这才第一次看爱丽丝，说："不要这样站着自言自语。告诉我，你的名字，你是干什么的？""我的名字是爱丽丝，然而……"

"多愚蠢的名字！它是什么意思？"矮胖子不耐烦地打断说。"难道名字一定要有意思吗？"爱丽丝疑惑地问。"当然要有啦，我的名字就是取意于我的形体。当然，这是一种很好、很漂亮的形体。而像你这样的名字，你可以成为任何形状了。"矮胖子说着，哼地笑了一声。

——刘易斯·卡洛尔，《爱丽丝镜中奇遇记》

输出正确的程序不一定是好程序。很显然，你希望自己的程序能够给出正确的输出，但这并不是全部。大多数程序会使用很多次，并会为了校正bug或适应用户新的需要而对某些地方进行修改。如果程序不好读，不好理解，就不容易修改，甚至无法进行有任何实际效果的修改。即使程序只会使用一次，也应该注意可读性问题。毕竟，在调试程序时是要阅读程

序的。

在本节中，我们将讨论4种可以使程序更具可读性的技术：有意义的名字、缩进、文档及已定义常量。

● **编程提示：为变量使用有意义的名字**

正如前面提到的，x和y这样的名字基本上永远都不会是好的变量名。变量的名字应该能暗示变量的用途。如果变量是对某种东西的计数，可以将其命名为count。如果变量表示税率，可以将其命名为taxRate。

除了为变量取一个有意义且编译器能接受的名字之外，还应该为其选择符合程序员使用习惯的名字。这样，在为一个有多个程序员参与的项目工作时，别人就更易读懂你编写的代码，并且更容易将它与别人编写的代码合并。按照惯例，变量名全部由字母和数字组成。如果名字中包含多个单词，就在单词的边界上用大写字母来“断开”它，如taxRate、numberOfTries和timeLeft。同样，就像我们刚刚给出的例子一样，用小写字母作为变量名的开始。以小写字母开头的习惯刚开始看起来可能有些奇怪，但这是一种广泛使用的惯例，你很快就会适应的。我们为其他一些项，如String这样的类名使用以大写字母开头的名字。

2.4.1　文档与注释

程序的文档用来说明程序是做什么的以及它是怎么做的。最好的程序是**自文档化**（self-documenting）的。这就是说，由于程序具有非常清晰的样式和经过精心挑选的变量名（和其他名字），阅读这个程序的任何一个程序员都可以很明显地看出这个程序是做什么的，以及它是怎么做的。要争取编写出这样的自文档化程序，但你的程序中还需要一些额外的解释，以使程序变得非常清晰。这些解释可以用注释的形式给出。

记住：编写自文档化的代码

自文档化的程序（或其他代码段）使用了经过精心挑选的变量名（和其他名字），具有非常清晰的样式，以至于即使程序中没有注释，阅读这个程序的任何一个程序员也能很明显地看出这个程序是做什么的，以及它是怎么做的。在可能的情况下，要努力使你所编写的程序成为自文档化的程序。

注释（comment）是写入程序中以帮助人们理解程序但会被编译器忽略的注解。在Java中，有两种插入注释的方法。第一种方法是在注释的起始处使用双斜杠（//），从双斜杠（//）之后开始到这行结束为止的所有内容都被当作注释处理，并会被编译器忽略。这种方法对短注释来说是很方便的，比如：

```
String sentence; //Spanish version
```

如果希望这种形式的注释跨越多行，那么，每行都必须在注释开始的地方包含双斜杠//。

第二种注释可以更容易地跨越多行，写在匹配符号对/*和*/之间的所有内容都是注释，并会被编译器忽略。例如：

```
/*
 This program should only
 be used on alternate Thursdays,
```

```
    except during leap years when it should
    only be used on alternate Tuesdays.
  */
```

这并不太像一个注释，但它确实说明了/*和*/的用法。

很多文本编辑器都会自动地用特殊的颜色显示注释，以强调它们。在本书中，我们也会用一种不同的方式来书写注释，如下面的注释所示：

```
  /**
    This program should only
    be used on alternate Thursdays,
    except during leap years when it should
    only be used on alternate Tuesdays.
  */
```

注意，这段注释在开头的/**中使用了两个而不是一个星号。要使其成为注释并不需要这样做，但当用一个名为javadoc的程序自动地从Java软件中提取文档时，就需要使用/**。我们将在本书稍后介绍javadoc，但现在就会开始使用双星号。

很难对什么时候应该插入注释，什么时候不应该插入注释进行解释。注释太多可能会和注释太少一样糟糕。注释太多，实际重要的信息就会被淹没在那些注释中，而这些注释只是描述了一些显而易见的内容。在向你展示更多Java特性的同时，我们会讲到哪些地方适合放置注释。现在，通常只需要两种类型的注释。

首先，在每个程序文件的开始处都应该有一条解释性的注释。这条注释应该给出与文件有关的所有重要信息：程序是干什么的，作者的名字是什么，如何联系作者，文件的最后修改日期是何时，如果文件中包含了课程作业的答案，还要说明作业号是多少。这条注释应该与图2-14顶端显示的注释类似。

需要使用的第二类注释是用来对任何不明显的细节进行解释的。比如，查看图2-14的程序。注意，程序中有两个名为radius和area的变量。很明显这两个变量会分别用来代表圆的直径和面积。那么，包含下面这样的注释就是一种错误的做法：

```
double radius;      //holds the radius of a circle.
```

但是，有些东西是不太明显的。直径使用的是什么单位？英寸？英尺？公尺？厘米？应该按下列形式添加一条对使用单位进行解释的注释：

```
double radius;      //in inches
double area;        //in square inches
```

图2-14中也显示了这两段注释。

```
import java.util.*;

/**
 Program to determine area of a circle.
 Author: Jane Q. Programmer.
 E-mail Address: janeq@somemachine.etc.etc.
 Programming Assignment 2.
 Last Changed: October 7, 2006.
 */
```

如果愿意，可以将这行代码放在下面一大段注释之后。

图2-14 注释与缩进

```
public class CircleCalculation
{
    public static void main(String[] args)
    {
        double radius; //in inches
        double area; //in square inches
        Scanner keyboard = new Scanner(System.in);

        System.out.println("Enter the radius of a circle in inches:");
        radius = keyboard.nextDouble();
        area = 3.14159 * radius * radius;
        System.out.println("A circle of radius " + radius + " inches");
        System.out.println("has an area of " + area + " square inches.");
    }
}
```

垂直线用来说明缩进的模式。

将在本章稍后给出此程序的改进版本。

屏幕对话示例

```
Enter the radius of a circle in inches:
2.5
A circle of radius 2.5 inches
has an area of 19.6349375 square inches.
```

图2-14　（续）

快速参考：Java注释

向Java程序（或一段Java代码）中添加注释的方式有两种：

（1）双斜杠//之后到这行结束之前所有的内容都是注释，会被编译器忽略。

（2）写在配对的符号对/*和*/之间的任何内容都是注释，会被编译器忽略。

2.4.2　缩进

程序中有很多结构。在较大的部分中会有一些较小的部分。例如，有的部分会以下面的形式开始：

```
public static void main(String[] args)
{
```

以一个闭花括号}作为结束。在这个部分内部，会有一些语句，比如赋值语句和System.out.println语句。如图2-14中的垂直线所示，在到目前为止我们见过的这类简单程序中，基本上都有三级嵌套结构。每级嵌套都应该缩进，以便更清晰地显示出嵌套结构。最外层的结构不用缩进，下一级嵌套结构要被缩进，在其内部的嵌套结构则要进行双重缩进。

这些嵌套结构通常是由花括号{}来标识的，但无论有没有花括号，都应该对每级嵌套内容进行缩进。

如果有的语句在一行写不下，可以将其写在两行或多行。但将一条语句书写在多行时，第二行及所有后续行的缩进要大于第一行的缩进。

我们比较喜欢为每级缩进使用4个空格。缩进更多，在这行为语句自身保留的空间就太少

了；而缩进少，则不能很好地显示缩进的结构。用两个或三个空格进行缩进并不是不合理，但我们发现使用4个空格进行缩进，结构最清晰。如果你在学习一门课程，应遵循老师指定的缩进规则。在一个编程项目中，可能会有一个样式表来说明应该缩进的空格数。在任何情况下，任意一个程序内部都应该保持缩进的一致性。

2.4.3 命名常量

再来看看图2-14中的程序，你可能会发现数字3.14159是π的近似值，在很多对圆的计算中都会用到这个数字，它通常会被写作π。但你可能无法确定3.14159是π而不是其他数字，有些人可能不知道数字3.14159从何而来。为了避免混淆，通常应该为3.14159这样的常量起一个名字，并在程序中使用这个名字，而不写出这个数字。比如，可以为数字3.14159起名为PI。然后，就可以将赋值语句

```
area = 3.14159 * radius * radius;
```

更清晰地写成

```
area = PI * radius * radius;
```

如何为一个数字，或其他的常量起一个像PI这样的名字呢？可以使用一个名为PI的变量，并将其初始化为你所期望的值，比如3.14159。但你可能会在无意中修改这个变量的值。Java提供了一种机制，允许定义并初始化一个变量，并且将这个变量的值固定，使其无法被改变。这种机制的语法为

```
public static final Type Variable = Constant;
```

例如，我们可以用下列方法将常量3.14159命名为PI：

```
public static final double PI = 3.14159;
```

可以只把这条语句当成将名字（如PI）赋给常量（如3.14159）的一种很长的、独特的方式，但我们可以对这一行语句的大部分内容进行解释。

```
double PI = 3.14159;
```

这一部分只是将PI声明为一个变量，并将其初始化为3.14159。这部分前面的单词用各种不同的方式对变量PI进行了修饰。单词public说明对名字PI的使用区域没有限制。单词static要到第5章再解释；现在，只要确保包含这个词即可。单词final意味着值3.14159是赋给PI的最终值，即不允许程序修改PI的值。

快速参考：命名常量

要为数字这样的常量命名，应将关键字public static final放在将常量作为初始值的变量声明前面。这个声明要放在类定义之内，而在main方法及任何其他方法定义之外（完整的示例请参见图2-15）。

语法：

```
public static final Type Variable = Constant;
```

举例：

```
public static final int MAX_STRIKES = 3;
public static final double MORTGAGE_INTEREST_RATE = 6.99;
public static final String MOTTO = "The customer is right!";
public static final char SCALE = 'K';
```

尽管不是必须这样做，但程序员通常习惯全部用大写字母来拼写命名常量。

```
import java.util.*;

/**
 Program to determine area of a circle.
 Author: Jane Q. Programmer.
 E-mail Address: janeq@somemachine.etc.etc.
 Programming Assignment 2.
 Last Changed: October 7, 2006.
*/

public class CircleCalculation2
{
    public static final double PI = 3.14159;

    public static void main(String[] args)
    {
        double radius; //in inches
        double area; //in square inches
        Scanner keyboard = new Scanner(System.in);

        System.out.println(
                 "Enter the radius of a circle in inches:");
        radius = keyboard.nextDouble();
        area = 3.14159 * radius * radius;
        System.out.println("A circle of radius " + radius + " inches");
        System.out.println("has an area of " + area + " square inches.");
    }
}
```

尽管将PI的定义放在这里不如放在上面清晰，但将它放在这里是合法的。

屏幕对话示例

```
Enter the radius of a circle in inches:
2.5
A circle of radius 2.5 inches
has an area of 19.6349735 square inches.
```

图2-15　命名一个常量

在图2-15中，我们重新编写了图2-14中的程序，用PI作为常量3.14159的定义。注意，PI的定义是放在程序的main部分之外的。正如在程序中指出的，常量的定义不是必须放在文件起始位置，但把它放在起始位置是个好习惯。这样，如果要修改命名常量定义，就会很方便。你可能不太想修改命名常量PI的定义，但可能会去修改其他程序中的其他一些命名常量的定义。比如，假设你有一个包含了下列已定义常量的金融程序：

```
public static final double MORTGAGE_INTEREST_RATE = 6.99;
```

假设利率变成了8.5%。你只要将已定义常量改成如下所示即可：

```
public static final double MORTGAGE_INTEREST_RATE = 8.5;
```

然后，要重新编译程序，但不需要对程序的其他内容进行任何修改了。

注意，如果要经常修改常量，MORTGAGE_INTEREST_RATE这样的已定义常量可以节省大

量的工作。要将抵押利率从6.99%改成8.5%，只需要修改一个数字即可。如果程序没有使用已定义常量，就要查找所有出现的6.99，并将其改成8.5。而且，即使这样也可能会出错。如果有时出现的6.99表示抵押利率，有些6.99表示其他类型的利率，则需确定每个6.99表示的是什么意思，那样肯定会造成一些混乱，可能还会引入错误。

注意，就像命名常量PI和MORTGAGE_INTEREST_RATE一样，我们拼写命名常量时用的都是（以下划线符号_作“分隔”的）大写字母，Java语言的定义并没有要求这样做，但这是一个几乎被普遍遵守的惯例，也是一个值得你接受的惯例。如果能很轻易地识别出变量和常量等内容，程序就会更易读一些。

自测题

30. Java中使用了哪两种类型的注释？
31. 下列Java代码会产生什么样的输出？

```
/**
Code for Exercise.
*/
System.out.println("One");
//System.out.println("Two");
System.out.println("And hit it!");
```

32. 美国有些州的立法机构通过了“修改π”值的法律。假设你住在这样一个州，法律规定π的值就是3.14。需要对图2-15中的程序进行什么样的修改才能使其符合那项法律呢？
33. 命名常量的常用拼写惯例是什么？

2.5 图形编程补充（选读）

在比赛真正结束之前，都不算结束。

——尤吉·贝拉，美国棒球名将

本节包含了第1章的一个applet，并根据本章讨论的样式规则对其进行了改写。但本节其余的所有内容都是关于JOPtionPane类的。JOptionPane类为你提供了一种对Java程序进行视窗I/O操作的方式。

第3章中也有少量关于JOptionPane（及其他图形材料）的内容。本章和第3章中所有关于JOptionPane的内容都是独立于本书其他内容的。无论你是否阅读了其他“图形编程补充”小节中的内容，都可以学习有关JOptionPane的内容。也可以略过有关JOptionPane的内容，去学习“图形编程补充”小节中的其他内容。

编程示例：应用于图形applet的样式规则

在图2-16中，改写了在图1-6中实现的applet程序。在这个版本中，有了帮助解释程序的注释，而且所有的整数实参都采用了已定义常量。乍一看，这些已定义常量好像只是使代码更复杂了，但是实际它们使修改代码的工作更简单了。

已定义常量中包含了一些约束条件，比如两个眼睛必须处于同一水平位置。这是由下列定义来保证的：

```
public static final int Y_LEFT_EYE = Y_RIGHT_EYE;
```

这样的程序通常需要通过调整各种整数实参的值来进行调试。使用已定义常量就可以更容易地找到这些值，比如嘴巴宽度的值。而且在改变嘴巴宽度时，改错数字的可能性也比较小。

```
import javax.swing.*;
import java.awt.*;

/**
 Applet that displays a happy face.
 Author: Jane Q. Programmer.
 E-mail Address: janeq@somemachine.etc.etc.
 Programming Assignment 3.
 Last Changed: October 9, 2006.
*/
public class HappyFace extends JApplet
{
    public static final int FACE_DIAMETER = 200;
    public static final int X_FACE = 100;
    public static final int Y_FACE = 50;

    public static final int EYE_WIDTH = 10;
    public static final int EYE_HEIGHT = 20;
    public static final int X_RIGHT_EYE = 155;
    public static final int Y_RIGHT_EYE = 95;
    public static final int X_LEFT_EYE = 230;
    public static final int Y_LEFT_EYE = Y_RIGHT_EYE;

    public static final int MOUTH_WIDTH = 100;
    public static final int MOUTH_HEIGHT = 50;
    public static final int X_MOUTH = 150;
    public static final int Y_MOUTH = 175;
    public static final int MOUTH_START_ANGLE = 180;
    public static final int MOUTH_DEGREES_SHOWN = 180;

    public void paint(Graphics canvas)
    {
        //Draw face outline:
        canvas.drawOval(X_FACE, Y_FACE, FACE_DIAMETER, FACE_DIAMETER);
        //Draw eyes:
        canvas.fillOval(X_RIGHT_EYE, Y_RIGHT_EYE, EYE_WIDTH, EYE_HEIGHT);
        canvas.fillOval(X_LEFT_EYE, Y_LEFT_EYE, EYE_WIDTH, EYE_HEIGHT);
        //Draw mouth:
        canvas.drawArc(X_MOUTH, Y_MOUTH, MOUTH_WIDTH, MOUTH_HEIGHT,
                       MOUTH_START_ANGLE, MOUTH_DEGREES_SHOWN);
    }
}
```

如果愿意，可以将这部分放在下面的大段注释之后。

产生的applet画面与图1-6产生的applet相同。

图2-16　用已定义常量和注释改写了的图1-6

2.5.1 JOptionPane

图2-17中包含了一个很简单的带有视窗界面的Java应用程序。在程序下面，显示了程序产生的3个窗口。这3个窗口一次只产生一个。出现的第一个窗口在图2-17中被标识为窗口1。用户在第一个窗口的文本框输入一个数字，然后点击OK按钮。点击OK按钮时，第一个窗口消失，第二个窗口出现。用类似的方法来处理第二个窗口，用户点击第二个窗口的OK按钮时，第二个窗口消失，第三个窗口出现。下面介绍这个过程中的一些细节。

```
import javax.swing.*;

public class JOptionPaneDemo
{
   public static void main(String[] args)
   {
      String appleString;
      appleString =
         JOptionPane.showInputDialog("Enter number of apples:");
      int appleCount;
      appleCount = Integer.parseInt(appleString);

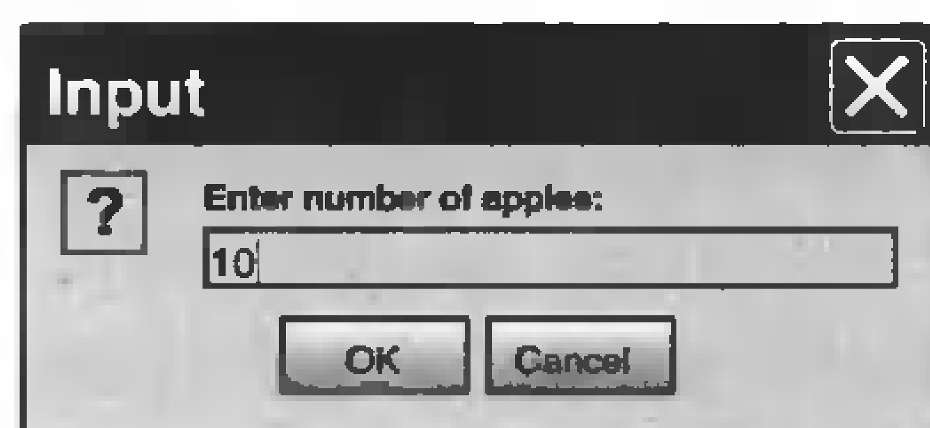
      String orangeString;
      orangeString =
         JOptionPane.showInputDialog("Enter number of oranges:");
      int orangeCount;
      orangeCount = Integer.parseInt(orangeString);

      int totalFruitCount;
      totalFruitCount = appleCount + orangeCount;

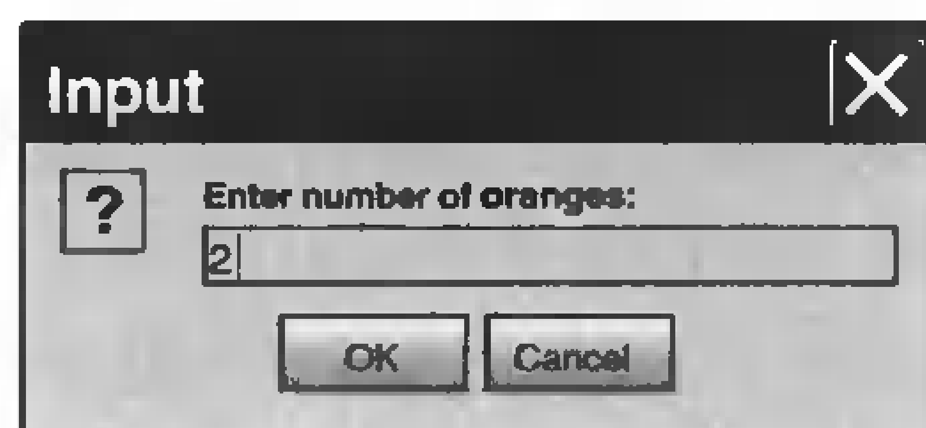
      JOptionPane.showMessageDialog(
         null, "The total number of fruits = " + totalFruitCount);

      System.exit(0);
   }
}
```

窗口1



窗口2



窗口3



点击OK按钮时，窗口消失，如果有下一个窗口，显示下一个窗口。

图2-17 带有JOptionPane I/O的程序

记住：运行JOptionPane程序

记住，运行如图2-17所示使用了JOptionPane的程序，其运行方式与运行任意一个应用程序的方式相同。运行时不用把它当成一个applet。

这个程序用JOptionPane类来构建与用户进行交互的窗口。程序的第一行代码告诉计算机到哪里去查找JOptionPane类的定义：

```
import javax.swing.*;
```

可以回想一下，我们提到过一个名为Swing的库，这是为视窗界面使用的一个类库。这些库被称为**包**（package），在Java程序中，Swing包是用javax.swing来表示的，其中的S是小写的。JOptionPane类就在这个包中。上面的代码行说明，你的程序可以使用包括JOptionPane类在内的javax.swing包中所有的类定义。要将这行代码放在所有使用JOptionPane类的程序文件的开头。

下面是给计算机的第一条程序指令。这条指令将appleString声明为一个String类型的变量：

```
String appleString;
```

接下来的两行启动了视窗动作。这两行是一条语句（指令）。一般除非将一条语句写在一行上会使那行代码太长、太不方便，否则通常应该将其写在一行上。这两行代码构成了一条的赋值语句：

```
appleString = JOptionPane.showInputDialog("Enter number of apples:");
```

JOptionPane是一个用来生成窗口的类，窗口可以用来获取输入，也可用来显示程序的输出。这是一个标准的预定义类，每次安装Java会自动安装它。只要使用我们在本节前面介绍过的import语句就能使JoptionPane可用。方法showInputDialog会产生一个用来获取输入的窗口。它会将字符串实参写入窗口以告知用户输入什么内容，在这个例子中，输入的就是“Enter number of apples:”。程序员根据自己希望得到什么样的输入来选择写入窗口的字符串。对方法showInputDialog的这次调用会生成图2-17中显示的第一个窗口。用户在文本框内点击鼠标，并输入一些内容。如果不喜欢所输入的内容，可以用退格键退回，并重新输入。一旦用户认可了输入的内容，就点击OK按钮，窗口就会消失了。作为替代，用户也可以按Enter（Return）键来取代点击OK按钮的动作。这由用户自己选择，但这些输入怎样才能到达你的程序呢？请继续往下读。

下面是方法调用：

```
JOptionPane.showInputDialog("Enter number of apples:");
```

它会返回（产生）用户在文本框中的输入。赋值语句的其余部分则说明了这些输入会传递到程序的什么地方。具体来说，输入字符串是存储在变量appleString中的。用JOptionPane实现输入操作时，只能输入字符串值。如果需要输入数字，必须由程序将输入字符串转换成数字。

下列行是另一个变量声明语句：

```
int appleCount;
```

int说明存储在变量appleCount中的数据必须是整数（integer）。编写这个程序的程序员希望用户在第一个输入窗口中输入一个整数，并希望程序将这个整数存储在变量appleCount中去。但程序已经将用户的输入作为字符串，而不是int类型的值存储起来了。假设用户输入了10，表示有10个苹果。用户可能认为他输入的是数字10。但用户实际输入的是字符'1'，后面跟着字符'0'，这就生成了字符串"10"。毕竟，用户输入"10"时所用的键盘和用户写入字符串"I

love the numeral 10"时所用的键盘是相同的，而且不论这个10表示的是字符串的一部分，还是用户所认为的数字，都是用相同的两个按键输入的。使用这些输入窗口时，必须明确的一点是：所有来自用户的程序输入（以及就此例来说传递给用户的所有输出）都是由字符串组成的。如果你希望程序将来自输入窗口的输入当作一个数字，程序就必须将"10"这样的字符串转换成数字，在这个例子中，就是转换成10。对计算机来说，"10"和10是有很大区别的。(在实际生活中，它们也是不同的，但我们通常忽略了这种不同。) "10"是一个由两个字符组成的字符串，而10是一个可以和其他数字进行加减的数字。用户输入的字符串（如"10"）是存储在变量appleString中的。程序要将存储在变量appleString中的字符串转换成int值，并将得到的int值存储在变量appleCount中。这是由下面这行程序代码实现的：

```
appleCount = Integer.parseInt(appleString);
```

Integer是一个类。parseInt是Integer类的一个方法。方法调用Integer.parseInt(appleString)会将存储在变量appleString中的字符串转换成相应的整数值。例如，如果存储在appleString中的字符串是"10"，那么这个方法调用就会返回整数10。变量名appleCount和等号表明这个数字10是存储在变量appleCount中的。

接下来的几行代码与刚才讨论过的内容只有少许差别。这几行代码生成了一个从用户获取输入字符串的输入窗口，并将相应的整数存储在变量orangeCount中。

```
String orangeString;
orangeString =
    JOptionPane.showInputDialog("Enter number of oranges:");
int orangeCount;
orangeCount = Integer.parseInt(orangeString);
```

前面几行代码产生的窗口即图2-17中所示的第二个窗口。

程序中接下来的两行代码使用的都是你已经了解了的Java语句，如下所示。

```
int totalFruitCount;
totalFruitCount = appleCount + orangeCount;
```

上面两行代码中的第一行说明totalFruitCount是一个int类型的变量，第二行代码是一条赋值语句，将totalFruitCount的值设置为appleCount和orangeCount这两个变量中的int值之和。

现在，程序要输出存储在变量totalFruitCount中的数字。输出是由下列语句实现的：

```
JOptionPane.showMessageDialog(
   null, "The total number of fruits = " + totalFruitCount);
```

showMessageDialog是JOptionPane类中的另一个方法。这个方法会显示一个可以展现某些输出的窗口。方法showMessageDialog有两个实参，由逗号分隔。现在，第一个实参总会被写成null，其含义要过段时间再解释。在此之前，把null当成一个由于我们还不需要任何“实际的”第一实参而使用的占位符号，也不会犯什么大错。第二个实参很好解释：它是写到输出窗口中的字符串。因此，前面的方法调用会生成图2-17中所示的第三个窗口。这个输出窗口会一直停留在屏幕上，直到用户点击OK按钮，或按Enter (Return) 键时，窗口才会消失。

注意，为JOptionPane的方法showMessageDialog指定输出字符串实参的方法与为System.out.println指定输出字符串实参的方法是一样的。需要使用存储在变量中的值时，通常都可以通过使用变量名来实现。因此，可以把totalFruitCount当作存储在totalFruitCount中的

整数值。而且，Java会自动地将存储在totalFruitCount中的整数值转换成相应的字符串。

下面显示的最后一条程序语句，只是用来说明程序应该结束了。

```
System.exit(0);
```

System是一个预定义的Java类，是由Java自动提供的，exit是System类中的一个方法。一调用方法exit，它就会终止程序。在我们要编写的程序中，整数实参0可以是任意整数，但按照传统，会使用0，因为0是用来说明程序的正常终止的。

快速参考：用于视窗I/O的JOptionPane

可以用方法showInputDialog和showMessageDialog为程序生成输入和输出窗口。使用这些方法时，要在含有程序的文件的起始处包含下列代码：

```
import javax.swing.*;
```

下面给出了使用这些方法的输入和输出语句的语法。

语法（输入）：

String_Variable = JOptionPane.showInputDialog(*String_Expression*);

举例：

```
String orangeString;
orangeString = JOptionPane.showInputDialog(
                              "Enter number of oranges:");
```

*String_Expression*会显示在一个窗口中，这个窗口包含了一个用户可以进行输入的文本框和一个OK按钮。在窗口中输入一个字符串并点击窗口中的OK按钮时，方法就会将输入的字符串返回，这样，用户输入的字符串就被存储在*String_Variable*中了。作为一种替代形式，按Enter（Return）键与点击OK按钮是等效的。注意，用这种方式实现输入时，所有的输入都是字符串输入。如果你希望用户输入的是整数等非字符串，程序就必须将表示数字的输入字符串转换成等价的数字。

语法（输出）：

JOptionPane.showMessageDialog(null, *String_Expression*);

举例：

```
JOptionPane.showMessageDialog(
    null, "The total number of fruits = " + totalFruitCount);
```

*String_Expression*会显示在一个窗口中，这个窗口包含了一个标记为OK的按钮。点击OK按钮或按Enter（Return）键时，窗口消失。

常见问题：为什么有些方法调用使用的是类名而不是调用对象？

通常，方法调用会用对象名作为调用对象，比如greeting.length()，其中greeting是一个String类型的变量。但当使用JOptionPane类的方法时，用类名JOptionPane取代了调用对象。这是怎么回事呢？有些特殊的方法不需要调用对象，是用类名来调用的。这些方法被称为**静态方法**，将在第5章中介绍。这些静态方法只占所有方法的一小部分，它们被用于某些基本任务，比如I/O。

▲ 易犯错误：输入不恰当内容的用户

当程序非正常终止时，称其**崩溃**（crash）了，这通常是因为什么地方出错了。用方法JOptionPane.showInputDialog（如图2-17所示）实现输入时，用户必须以正确的格式输入，否则

程序就可能会崩溃。在Java中整数数字是不能包含逗号的，所以，如果程序期望用户输入一个整数，而用户输入了2,000，程序就会崩溃（用户应该输入2000）。稍后将介绍如何编写出更健壮的视窗程序，而不需要用户那么博学，那么小心。在此之前，你只能告诉用户小心一些。 △

▲ 易犯错误：忘记了System.exit(0);

如果忽略了图2-17程序中的最后一行，即

```
System.exit(0);
```

所有的事情都会像我们描述的那样工作。用户用输入窗口进行输入，输出窗口会显示输出。点击输出窗口中的OK按钮时，输出窗口会消失，但是程序不会终止。窗口会消失，但“不可见的”程序还在那里。它会挂在那里，占用计算机的资源，可能还会妨碍你做其他事情。因此，在所有的视窗程序中都不要忘记System.exit(0);。

但是，如果忘记了System.exit(0);程序自己无法结束，该怎么办？终止一个自己未结束的程序的方法取决于你使用的操作系统。在很多（但不是所有的）系统中，都可以通过按Ctrl+C快捷键来结束一个程序。

编写（除了applet之外）带有视窗界面的程序时，通常都要用下面的语句来终止程序：

```
System.exit(0);
```

如果程序中没有使用视窗界面，比如2.1节~2.4节中的程序，就不需要调用System. exit(0);。 △

▲ 易犯错误：只输出一个数字

在图2-17中，最终的程序输出是通过下列语句发送到输出窗口中去的：

```
JOptionPane.showMessageDialog(
     null, "The total number of fruits = " + totalFruitCount);
```

为每条输出都打上相应的标记是一种很好的编程风格。因此，下列字符串对程序的样式和可理解性来说是很重要的：

```
"The total number of fruits = "
```

而且，如果输出中不包含字符串，这个方法调用甚至都不能编译。例如，下列代码就不能编译：

```
JOptionPane.showMessageDialog(null, totalFruitCount);
```

方法showMessageDialog不会接受一个int值（或任何其他基本类型的值）作为它的第二个实参。如果用加号将一个变量或数字与一个字符拼接接起来，实参就会被转换成字符串，这样才会接受实参。 △

常见问题：为什么有些程序需要使用System.exit，而有些程序不需要呢？

如图2-17所示使用了视窗界面的非applet程序，要求以下列方法调用作为程序的结束：

```
System.exit(0);
```

而像本章前几节中的程序那样，使用简单的文本输入和输出的程序则不需要这个方法调用。为什么会有这种区别呢？

使用简单文本输入和输出的程序不需要System.exit方法调用的原因是Java可以很容易地分辨出程序什么时候应该结束；所有的语句都执行完毕，程序就该结束了。

applet也有一种内建机制来结束applet程序。当显示applet的Web页面跳转了，或applet浏览器窗口关闭了的时候，applet就终止了。对那些带有视窗界面的非applet程序来说，情况就不那么简单了。

图2-17中所示的带有视窗界面的程序确实需要调用方法System.exit的原因是：Java无法轻易地分辨出视窗程序应该在什么时候结束。很多视窗程序都只有在某些按键被点击，或执行了某些其他动作时才会结束，而这些细节是由程序员而不是Java语言决定的。对迄今为止所见过的这些简单程序来说，当程序中所有的语句都执行完毕时，程序确实碰巧结束了；但对那些更复杂的视窗程序来说，程序的结束位置不是那么容易找到的．因此，就必须在程序应该结束的地方插入一个对System.exit的调用，告诉Java程序什么时候会结束。

自测题

34. 在下面两行代码中，有一个标识符（单词）命名了一个类，有一个标识符命名了一个方法，还有一些是实参。类名是什么？方法名是什么？实参是什么？
```
appleString = JOptionPane.showInputDialog("Enter number of apples: ");
```
35. 给出一条Java语句，使其在屏幕上显示一个带有下列消息的窗口：
```
I Love You.
```
36. 给出一条一执行就会将程序终止的Java语句。
37. 如果在图2-17的程序中省略了下列方法调用，会发生什么情况？程序会编译吗？程序能够没有任何问题地运行吗？
```
System.exit(0);
```
38. 编写一个完整的Java程序，使其生成一个带有消息Hello World！的窗口。程序其他什么事也不做。
39. 编写一个完整的Java程序，使其在运行时完成下列任务：程序显示一个输入窗口，请求用户输入一个整数。当用户输入一个整数并点击OK按钮时，这个输入窗口消失，并出现一个输出窗口。输出窗口只是告诉用户他输入了什么数字。点击输出窗口中的OK按钮时，程序结束。

2.5.2 输入其他数字类型

可以用方法JOptionPane.showInputDialog从用户处获取任意数字类型的输入。JOptionPane.showInputDialog的用法与我们用它来获取整数（即获取int类型的输入）的方法完全相同。但是如果你希望获得除int之外其他类型的数字，就要使用不同于Integer.parseInt的方法，将输入字符串转换成一个数字。例如，下列代码请求用户输入一个double类型的值，并将其存储在double类型的变量decimalNumber中。

```
String numberString;
numberString = JOptionPane.showInputDialog(
                "Enter a number with a decimal point:");
double decimalNumber;
decimalNumber = Double.parseDouble(numberString);
```

图2-18列出了每种数字类型对应的正确转换方法。

类型名	转换方法
byte	Byte.parseByte(*String_To_Convert*)
short	Short.parseShort(*String_To_Convert*)
int	Integer.parseInt(*String_To_Convert*)
long	Long.parseLong(*String_To_Convert*)
float	Float.parseFloat(*String_To_Convert*)
double	Double.parseDouble(*String_To_Convert*)

图2-18　将字符串转换成数字的方法

可以用第二列给出的方法将一个String类型的值转换成第一列给出的类型的值。第二列中的每个方法都会返回一个对应于第一列中类型的值。*String_To_Convert*必须是第一列给出类型值的正确的字符串表达。例如，要转换成一个int类型的值，*String_To_Convert*就必须是以通常方式书写的、不含任何小数点的（在int类型范围之内的）整数。

■ Java提示：多行输出窗口

如果想用方法JOptionPane.showMessageDialog输出多行文本，可以在作为第二个实参使用的字符串中插入换行符'\n'。如果字符串太长（在多行输出时经常会出现这种情况），可以将每行都写成一个单独的、'\n'结束的字符串，并用加号将各行连接起来。如果行很长，或者有很多行，窗口就会变得更大一些，以包含所有的输出。

例如：

```
JOptionPane.showMessageDialog(null,
                              "The number of apples\n"
                            + "plus the number of oranges\n"
                            + "is equal to " + totalFruit);
```

假设totalFruit是一个值为12的int类型变量，JOptionPane.showMessageDialog调用就会生成图2-19所示的窗口。

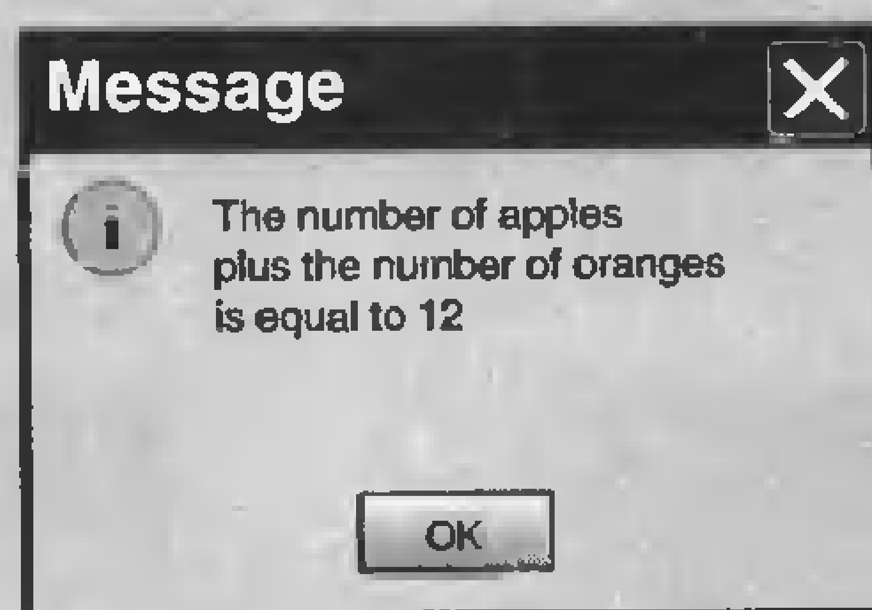


图2-19 多行输出窗口

编程示例：带有视窗I/O的换零钱程序

图2-20的程序中使用了一个多行输入窗口和一个多行输出窗口。除了输入和输出之外，这个程序与图2-6中的程序是一样的。如果对计算硬币数的细节有什么不太清楚的地方，可以回过头去看看对图2-6的解释。

```
import javax.swing.*;

public class ChangeMakerWindow
{
    public static void main(String[] args)
    {
        String amountString =
                 JOptionPane.showInputDialog(
                       "Enter a whole number from 1 to 99.\n"
                     + "I will output a combination of coins\n"
                     + "that equals that amount of change.");
```

图2-20 带有I/O窗口的换零钱程序

```
        int amount, originalAmount, quarters, dimes, nickels, pennies;

        amount = Integer.parseInt(amountString);
        originalAmount = amount;

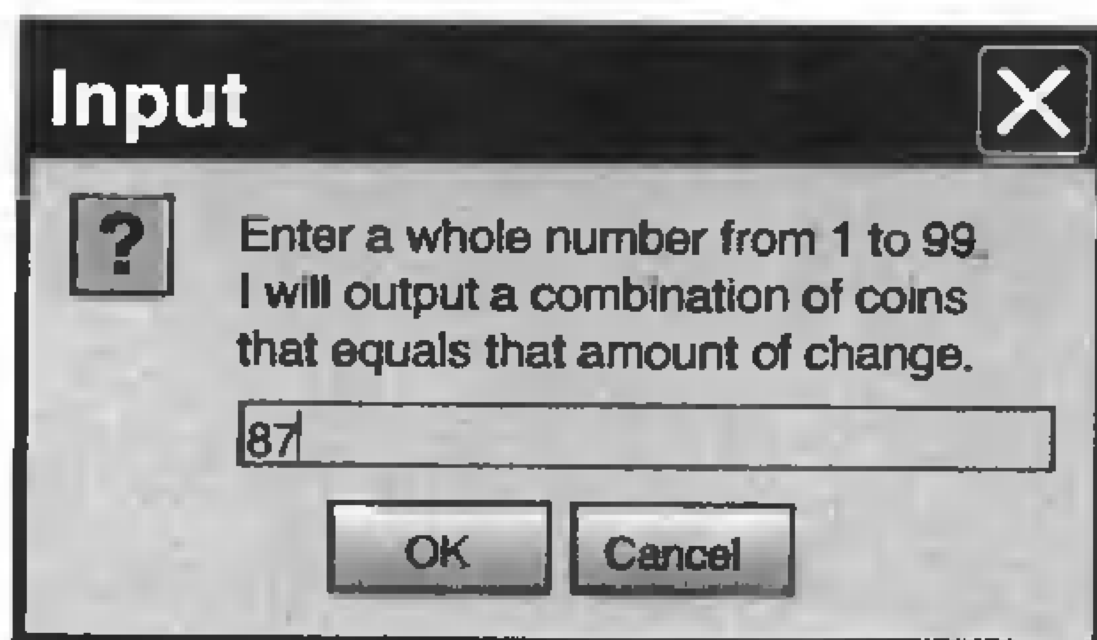
        quarters = amount/25;
        amount = amount%25;
        dimes = amount/10;
        amount = amount%10;
        nickels = amount/5;
        amount = amount%5;
        pennies = amount;

        JOptionPane.showMessageDialog(null,
                        originalAmount
                      + " cents in coins can be given as:\n"
                      + quarters + " quarters\n"
                      + dimes + " dimes\n"
                      + nickels + " nickels and\n"
                      + pennies + " pennies");

        System.exit(0);
    }
}
```

不要忘记在带有输入或输出窗口的程序中需要使用System.exit。

输入窗口



输出窗口

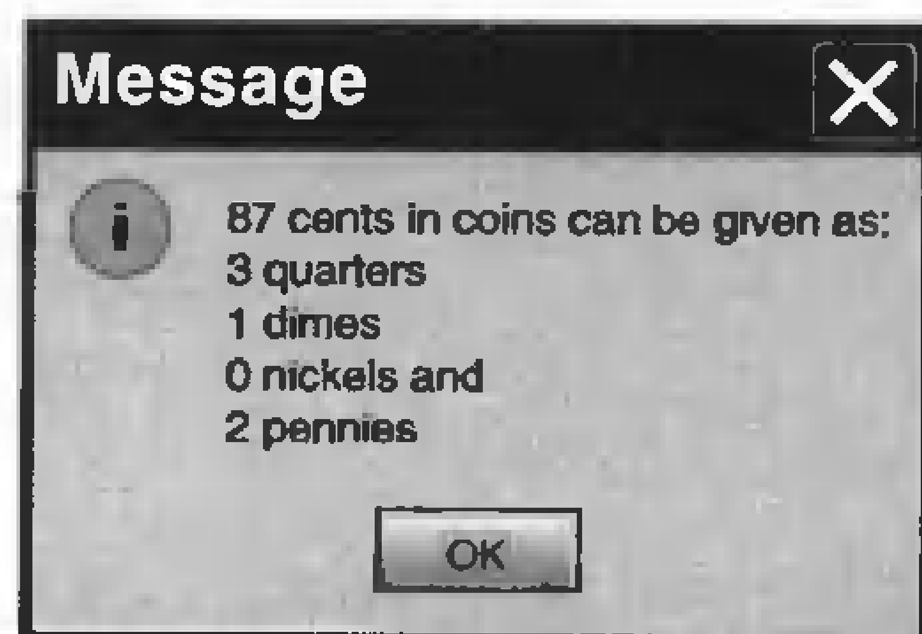


图2-20　（续）

一定要注意，输入窗口和输出窗口都显示了多行文本。

注意，我们没有忘记import语句：

```
import javax.swing.*;
```

也没有忘记对一个使用了JOptionPane的程序来说，需要调用System.exit方法。如果忘记了import语句，编译器会报警。但是，如果忽略了对System.exit方法的调用，编译器不会报错，但程序也不会终止，甚至在所有的语句都执行完，所有的窗口都消失了之后，程序也不会终止。对视窗程序来说，*在比赛真正结束之前，都不算结束*。真正结束了程序的是System.exit，而不是没有语句可执行了。

小　结

■ 变量可以用来存储数字这样的值。变量的类型必须与存储在变量中的值的类型相匹配。

- 应该为变量（以及程序中所有其他项）起一个能够说明这个变量用途的名字。
- 所有变量都应该在程序使用变量的值之前进行初始化。
- 算术表达式中的圆括号指出了操作执行的顺序。
- 可以用Scanner类中的方法来读取键盘输入。
- 期望用户从键盘上输入数据时，程序应该输出一个提示行。
- 可以使用String类型的变量和常量。String是一种和基本类型很相似的类类型。
- 可以用加号来拼接两个字符串。
- String类中有一些可以用来进行字符串处理的方法。
- 应该为程序中的数值常量命名，并在程序中使用这些名字而不要直接写出相应数值。
- 程序应该尽可能地实现自文档化。但也应该插入一些注释，对不太清楚的问题进行解释。
- 可以用JOptionPane类来实现使用了窗口的I/O。[1]

✓ 自测题答案

1. 下面这些都是合法的变量名：

   rate1、TimeLimit、numberOfWindows

   但TimeLimit违反了变量应该以小写字母开头的惯例，最好不用。更好的选择应该是timeLimit。1stPlayer是非法的，因为它是以数字开头的。myprogram.java是非法的，因为它包含了一个非法字符——点。最后，由于long是个关键字，所以用它作为变量名也是非法的。
2. 是的，一个Java程序中可以有两个名为aVariable和avariable的不同变量，因为aVariable和avariable的大小写不同，所以在Java中它们是不同的标识符。但是，最好不要使用仅仅在大小写形式上有所不同的标识符。
3. `int count = 0;`
4. `double rate = 0.0, time = 0.0;`

   下列代码也是正确的，因为Java会自动地将int值0转换成double值0.0：

   `double rate = 0, time = 0;`
5.
```
int miles = 0;
double flowRate = 50.56;
```
6. `interest = 0.05 * balance;`

   下列代码也是正确的：

   `interest = balance * 0.05;`
7. `interest = balance * rate;`
8. `count = count + 3;`
9.
```
b
c
c
```
   由于最后一条赋值语句a = b;中没有使用引号，所以最后一条语句输出的是c。最后一条赋值语句设置了变量a的值，使其等于变量b的值，变量b的值就是'c'。

① 2.5节中介绍的内容。

10. `(int)symbol - (int)'0'`

　　要看明白这种方法确实有效，首先要注意到它对'0'是有效的，然后看到它对'1'是有效的，然后是'2'，依次类推。可以用实际的数字来取代(int)'0'，但使用(int)'0'会更好理解一些。

11. `result is 30`

12.
```
quotient = 2
remainder = 1
```

13. `(1/2) * 2 is equal to 0.0`

　　这是因为1/2是个整数除法，它会丢弃小数点后面的内容，生成0而不是0.5。

14. 对话会变成下列形式：

```
Enter a whole number from 1 to 99
I will output a combination of coins
that equals that amount of change.
87
2 cents in coins can be given as:
3 quarters
1 dimes
0 nickels and
2 pennies
```

15. `result is 5`

16.
```
n == 3
n == 2
```

17. `How do you doSeven of Nine.`

　　注意，doSeven中间没有空格。

18.
```
7

b
```

19. `defg`

20.
```
abc
def
```

21. `abc\ndef`

22. `HELLO JOHN`

23. `End`

　　注意，这两个字符串是不相等的。

24.
```
Equal
End
```

25.
```
System.out.println("Once upon a time,");
System.out.println("there were three little programmers. ");
```

　　因为我们没有说明接下来的输出应该是什么样的，所以下列语句也是正确的：

```
System.out.println("Once upon a time,");
System.out.print("there were three little programmers.");
```

26. 使用System.out.println，接下来的输出会显示在下一行上。（接下来的输出是指在所讨论的System.out.println之后的第一条输出语句产生的输出。）使用System.out.print，接下来的输出会显示在同一行上。

27.
```
import java.util.*;
public class FirstScannerExercise
```

```
{
     public static void main(String[] args)
     {
             Scanner keyboard = new Scanner(System.in);
             int n1, n2;
             System.out.println("Enter two whole numbers:");
             n1 = keyboard.nextInt();
             n2 = keyboard.nextInt();
             System.out.println("You entered: " + n1 + "  " + n2);
     }
}
```

28.
```
import java.util.*;
public class SecondScannerExercise
{
     public static void main(String[] args)
     {
         Scanner keyboard = new Scanner(System.in);
         String word1, word2, word3;
         System.out.println("Enter a line of text with three words:");
         word1 = keyboard.next();
         word2 = keyboard.next();
         word3 = keyboard.next();
         System.out.println("With space corrected, you entered:");
         System.out.println(word1 + "  " + word2 + "  " + word3);
     }
}
```

29. 用加号连接起来的3个字符串中的第二个是空字符串。因此，输出为

```
HelloJoe
```

30. 两种注释类型为//注释和/* */注释。在同一行上跟在//后面的所有内容都是注释。在/*和配对的*/之间的所有内容都是注释。

31.
```
One
And hit it!
```

32. 将代码行

```
public static final double PI = 3.14159;
```

改成

```
public static final double PI = 3.14;
```

由于double类型的值是以有限的精确度存储的，所以你可以说这也不是“精确的”3.14，但对任何会用立法来规定π值的立法者来说，都不大可能明白这种微妙的差别。

33. 程序员的惯例是拼写命名常量时全部使用大写字母，并用下划线来分隔单词。

34. JOptionPane是类，showInputDialog是方法，"Enter number of apples: "是实参。

35. `JOptionPane.showMessageDialog(null, "I Love You.");`

36. `System.exit(0);`

37. 程序会编译，会运行，看起来甚至会没有任何问题地运行。但是，即使在关闭了所有窗口之后，程序仍然在运行。它会继续运行，并耗费资源，而Java代码中没有任何语句会将程序终止。你不得不用操作系统来终止程序。

38. `import javax.swing.*;`

```
public class ExerciseHello
{
    public static void main(String[] args)
    {
        JOptionPane.showMessageDialog(null, "Hello World!");
        System.exit(0);
    }
}
```

39. 你可能希望将numberString转换成一个int类型，但这对用户在程序运行时看到的内容没有任何影响。

```
import javax.swing.*;
public class ExerciseEchoNumber
{
    public static void main(String[] args)
    {
        String numberString;
        numberString=
        JOptionPane.showInputDialog("Enter a whole number: ");
        JOptionPane.showMessageDialog(
          null, "The number is " + numberString);
        Sytem.exit(0);
    }
}
```

编程项目

没有为包含JOptionPane的2.5节单独设置的编程项目。下列任何一个编程项目都可以用JOptionPane实现，也可以不用JOptionPane实现。

1. 编写一个程序，读入3个整数，输出这3个整数的平均值。
2. 编写一个程序，读入每月的抵押贷款数额和未达余额（即仍然亏欠的数额），然后输出作为利息支付的数额和作为本金支付的数额（即用来减少债务的款项数额）。假设年利率为7.49%。为利率使用一个已定义常量。注意，支付是每月进行的，所以利息只有年利率7.49%的1/12。
3. 编写一个程序，读入一个4位数（如1998），然后将这个数输出，每行输出一个数字，就像下面的形式：

```
1
9
9
8
```

   程序提示应该告诉用户输入一个四位数，然后假定用户会遵循这条指示。
4. 编写一个程序，读入一行文本，将第一次出现的"hate"改成"love"，然后将此行输出。例如：

```
Enter a line of text.
I hate you.
I have rephrased that line to read:
I love you.
```

   可以假定在输入中会出现单词"hate"。如果在这一行中单词"hate"出现了多次，程序只会对第一次

出现的"hate"进行替换。

5. 编写一个程序，读取一行文本作为输入，然后将第一个单词移到这行的末尾，并将此行输出。例如：

```
Enter a line of text. No punctuation please.
Java is the language.
I have rephrased that line to read:
Is the language Java
```

假设在第一个单词之前没有空格，并且第一个单词的结束是由一个空格（而不是逗号或其他标点）来标识的。

6. 编写一个程序，打印出投掷8次硬币的统计信息。对这8次投掷，可以输入'h'来表示正面，也可以输入't'来表示反面。然后，程序会将正反面的总数和正反面所占的比例打印出来。用递增运算符对输入的每个'h'和't'进行计数。例如：

```
For each coin toss enter either 'h' for heads
or 't' for tails.

First toss: h
Second toss: t
Third toss: t
Fourth toss: h
Fifth toss: t
Sixth toss: h
Seventh toss: t
Eighth toss: t

Number of heads: 3
Number of tails: 5
Percent heads: 37.5
Percent tails: 62.5
```

7. 编写一个程序，请求用户输入一个朋友或亲戚的名字、一种喜欢的颜色、一种喜欢的食物和一种喜欢的动物，然后以用户的输入取代下两行中的斜体项，并将这两行输出：

I had a dream that *Name* ate a *Color Animal*
and said it tasted like *Food*!

例如，如果用户输入Jake作为人名，blue作为颜色，hamburger作为食物，dog作为动物，输出就是：

```
I had a dream that Jake ate a blue dog
and said it tasted like hamburger!
```

不要忘了在结尾加上感叹号。

8. 编写一个程序，用下列公式将华氏度转换成摄氏度：

DegreesC = 5(*DegreesF* - 32)/9

提示用户输入一个以华氏度表示的温度（只能是整数的温度，不包含小数部分），然后让程序打印出等价的摄氏温度，其中要包含至少到小数点后一位的分数部分。例如：

```
Enter a temperature in degrees Fahrenheit: 72
72 degrees Fahrenheit = 22.2 degrees Celsius.
```

9. 编写一个程序，用来确定售货机找出的零钱数。售货机中的每样物品价值25美分至1美元，以5美分递

增（25，30，35，…，90，95或100），为物品付款时售货机只接受1美元的钞票。例如：

```
Enter price of item
(from 25 cents to a dollar, in 5-cent increments): 45
You bought an item for 45 cents and gave me a dollar,
so your change is
2 quarters,
0 dimes, and
1 nickel.
```

第3章

流 程 控 制

“请你告诉我，离开这里应该走哪条路？”

“这主要取决于你想上哪儿去”，猫说。

——刘易斯·卡洛尔，《爱丽丝漫游奇境记》

流程控制（flow of control）是程序执行动作的顺序。到本章为止，程序的执行顺序都很简单。动作都是按照其书写顺序执行的。本章将介绍如何编写一些流程控制较复杂的程序。Java和大多数其他编程语言，都使用了两种语句来规范控制流程：**分支语句**（branching statement）会从两个或多个可能的动作列表中选择一个动作；**循环语句**（loop statement）会不断地重复一个动作，直到满足某种停止条件为止。

目标

- 了解Java分支语句。
- 了解循环的概念。
- 了解boolean类型。
- 选读，了解如何在图形程序中使用颜色及其他增强功能。

预备知识

在阅读本章之前要熟悉第2章中除了2.5节之外的所有内容。

3.1 分支语句

在路上遇到岔口时，接受它。

——尤吉·贝拉

对分支语句的讨论，将以一种能在两种可能的备选动作中进行选择的Java语句作为开始。

3.1.1 if-else语句

与在日常生活中一样，在程序中事情有时也会向两种不同的方向发展。对活期存款账户来说，如果活期存款账户中还有钱，银行就会付给你一点儿利息；另一方面，如果活期存款账户已经透支，账户余额为负，银行就要向你收取一定的罚金，这会使你的账户余额比原来负得更多。可以通过下列名为if-else语句的Java语句，将这一点反映到银行的账务程序中：

```
if (balance >= 0)
```

```
    balance = balance + (INTEREST_RATE * balance)/12;
else
    balance = balance - OVERDRAWN_PENALTY;
```

因为键盘上没有符号≥，所以在Java中用符号组>=来表示大于或等于。

if-else语句的含义实际上与它作为一条英文句子的句意相同。程序执行一条if-else语句时，首先会检查if后面圆括号中的表达式。表达式或者为真，或者为假。如果为真，就执行else之前的语句；如果为假，就执行else之后的语句。在前面的例子中，如果balance为正（或零），就执行下列动作（因为计算的只是12个月中的1个月，所以要除以12）：

```
balance = balance + (INTEREST_RATE * balance)/12;
```

另一方面，如果balance的值为负，就执行下列动作：

```
balance = balance - OVERDRAWN_PENALTY;
```

图3-1在一个完整的程序中显示了这条if-else语句。

```
import java.util.*;

public class BankBalance
{
    public static final double OVERDRAWN_PENALTY = 8.00;
    public static final double INTEREST_RATE = 0.02;//2% annually

    public static void main(String[] args)
    {
        double balance;
        System.out.print("Enter your checking account balance: $");
        Scanner keyboard =  new Scanner(System.in);
        balance = keyboard.nextDouble();
        System.out.println("Original balance $" + balance);
        if (balance >= 0)
            balance = balance + (INTEREST_RATE * balance)/12;
        else
            balance = balance - OVERDRAWN_PENALTY;
        System.out.println("After adjusting for one month");
        System.out.println("of interest and penalties,");
        System.out.println("your new balance is $" + balance);
    }
}
```

屏幕对话示例1

```
Enter your checking account balance:$505.67
Original balance $505.67
After adjusting for one month
of interest and penalties,
your new balance is $506.51278
```

屏幕对话示例2

```
Enter your checking account balance:$-15.53
Original balance $-15.53
After adjusting for one month
of interest and penalties,
your new balance is $-23.53
```

图3-1　使用if-else语句的程序

如果想在每条分支中包含多条语句，只要将这些语句包含在一对花括号中即可，如下例所示：

```
if (balance >= 0)
{
    System.out.println("Good for you. You earned interest.");
    balance = balance + (INTEREST_RATE * balance)/12;
}
else
{
    System.out.println("You will be charged a penalty.");
    balance = balance - OVERDRAWN_PENALTY;
}
```

如果省略了else部分，当if后面的表达式为假时就不会发生任何事情了。例如，如果银行不收取任何透支罚金，这个程序中的语句就会如下所示：

```
if (balance >= 0)
{
    System.out.println("Good for you. You earned interest.");
    balance = balance + (INTEREST_RATE * balance)/12;
}
```

想看明白这条语句是如何工作的，要添加一些额外的语句使其具有更多的上下文信息，如下所示（keyboard是一个常见的Scanner对象）。

```
System.out.print("Enter your balance $");
balance = keyboard,nextDouble();
if (balance >= 0)
{
    System.out.println("Good for you. You earned interest.");
    balance = balance + (INTEREST_RATE * balance)/12;
}
System.out.println("Your new balance is $" + balance);
```

现在假设你的活期存款账户余额为100.00美元，那么，对话可能就是这样的：

```
Enter your balance $100.00
Good for you. You earned interest.
Your new balance is $100.16
```

if之后的表达式为真，因此可以得到一点儿利息。（如图3-1所示，我们使用的利率是每年2%，但在这个例子中，你只要注意到添加了一些利息就行了，利息的确切数额是无关紧要的。）

接下来，假设你的余额为$-50.00（即-50美元）。那么，对话可能就是这样的：

```
Enter your balance $-50.00
Your new balance is $-50.00
```

在这种情况下，if之后的表达式为假，而且没有else部分，因此，什么也没发生——余额没有发生变化，程序只是进入到下一条输出语句。

快速参考：if-else语句

下面提到的*Boolean_Expression*是一个为真或为假的表达式，比如balance <= 0。

语法（基本形式）：

```
if (Boolean_Expression)
    Statement_1
else
```

Statement_2

如果*Boolean_Expression*为真，就执行*Statement_1*；否则，就执行*Statement_2*。

举例：

```
if (time < limit)
    System.out.println("You made it.");
else
    System.out.println("You missed the deadline.");
```

语法（省略else部分）：

```
if (Boolean_Expression)
    Action_Statement
```

如果*Boolean_Expression*为真，就执行*Action_Statement*；否则，什么也不会发生，程序接着执行下一条语句。

举例：

```
if (weight > ideal)
    calorieAllotment = calorieAllotment - 500;
```

多条语句的选择：

如果要在一个选择中包含多条语句，就要像下面的例子这样用花括号将它们组织起来：

```
if (balance >= 0)
{
    System.out.println("Good for you. You earned interest.");
    balance = balance + (INTEREST_RATE * balance)/12;
}
else
{
    System.out.println("You will be charged a penalty.");
    balance = balance - OVERDRAWN_PENALTY;
}
```

3.1.2 布尔表达式简介

布尔表达式（boolean expression）是一个只能取真或假这两个值的表达式。布尔这个名字来自于George Boole，他是19世纪英国的逻辑学家和数学家，他所做的工作与这类表达式有关。

我们已经在`if-else`语句中使用过简单的布尔表达式了。最简单的布尔表达式就是对两个表达式的比较，例如：

```
time < limit
```

以及

```
balance <= 0
```

注意，布尔表达式并不是一定要包含在圆括号中。但是，在将布尔表达式用于`if-else`语句时，确实需要将其包含在圆括号中。

图3-2显示了各种可以用来对两个表达式进行比较的Java比较运算符。

可以用Java版本的“与”将表达式连接起来，以便用较简单的布尔表达式生成较复杂的布尔表达式。Java版本的“与”写作`&&`。例如，考虑下列代码：

```
if ((pressure > min) && (pressure < max))
    System.out.println("Pressure is OK.");
```

```
else
    System.out.println("Warning: Pressure is out of range.");
```

数学表示法	名　称	Java表示法	Java示例
=	等于	==	balance == 0　answer == 'y'
≠	不等于	!=	income != tax　answer != 'y'
>	大于	>	expenses > income
≥	大于或等于	>=	points >= 60
<	小于	<	pressure < max
≤	小于或等于	<=	expenses <= income

图3-2 Java比较运算符

如果pressure的值大于min，而且（and）pressure的值小于max，那么输出如下：

```
Pressure is OK.
```

否则，输出如下：

```
Warning: Pressure is out of range.
```

注意，在Java中不能使用下面这样的不等式串：

```
min < pressure < max
```

作为替代，必须按下列形式将每个不等式单独表示出来，并用&&将它们连接起来：

```
(pressure > min) && (pressure < max)
```

用&&将两个较小的表达式连接起来形成一个较大的布尔表达式时，只有两个较小的表达式都为真，较大的表达式才为真。如果至少有一个较小的表达式为假，较大的表达式就为假。例如：

```
(pressure > min) && (pressure < max)
```

只有在(pressure > min)和(pressure < max)都为真时，才为真；否则表达式就为假。

记住：用&&进行“与”运算

在Java中用&&这对符号来表示“与”。通过&&，可以用两个较小的布尔表达式构成一个较大的布尔表达式。

语法：

(*Sub_Expression_1*) && (*Sub_Expression_2*)

举例：

```
if ((pressure > min) && (pressure < max))
    System.out.println("Pressure is OK.");
else
    System.out.println("Warning: Pressure is out of range.");
```

Java表示“或”的方式为||，可以通过输入两条竖线来创建这个符号。(在某些系统中，符号|会在行上打印出一个间隔。）可以通过||用较小的布尔表达式构成一个较大的布尔表达式。它的含义本质上与英文单词“or”是一样的。例如，考虑下列代码：

```
if ((salary > expenses) || (savings > expenses))
    System.out.println("Solvent");
else
```

```
    System.out.println("Bankrupt");
```

如果salary的值大于expenses的值，或者savings的值大于expenses的值，或者两者都为真，就会输出Solvent；否则，就会输出Bankrupt。注意，如果||连接的两个表达式都为真，那么整个表达式也为真。有时也会将其称为同或（inclusive or）。

记住：用||进行“或”运算

在Java中用||这对符号来表示“或”。通过||，可以用两个较小的布尔表达式构成一个较大的布尔表达式。

语法：

(*Sub_Expression_1*) || (*Sub_Expression_2*)

举例：

```
if ((salary > expenses) || (savings > expenses))
    System.out.println("Solvent");
else
    System.out.println("Bankrupt");
```

正如前面指出的那样，if-else语句中的布尔表达式必须用圆括号括起来。使用了&&运算符的if-else语句通常是按如下方式使用括号的：

```
if ((pressure > min) && (pressure < max))
    System.out.println("Pressure is OK.");
else
    System.out.println("Warning: Pressure is out of range.");
```

(pressure > min)中的圆括号和(pressure < max)中的圆括号都不是必需的。不过为了提高可读性，通常都会使用这些括号。

在包含||的表达式中使用圆括号的方法与在包含&&的表达式中使用圆括号的方法一样。

在Java中，可以用!对布尔表达式求反。例如：

```
if (!(number >= min))
    System.out.println("Too small");
else
    System.out.println("OK");
```

如果number不大于或等于min，输出Too small；否则，输出OK。

通常可以（也应该）避免使用!。例如，前面的if-else语句等效于下列语句：

```
if (number < min)
    System.out.println("Too small");
else
    System.out.println("OK");
```

如果不使用!，程序就会更容易理解一些。

▲ 易犯错误：对字符串使用==

尽管==能够正确地测试两个基本类型（如两个数字）的值是否相等，但将其应用于对象时，就有了不同的含义[①]。回想一下，对象就是某个类（如一个字符串类）的成员。所有字符串都属于String类，

① 将==应用于两个字符串（或任意两个对象）时，测试的是它们是否存储在相同的存储单元中，但在第4章之前不会进行那么深入的讨论。到目前为止，只需要注意，==没有测试两个字符串是否相等，而是做了其他一些事情。

因此，将==应用于两个字符串时，它并没有对这两个字符串是否相等进行测试。要想测试两个字符串（或者任意两个对象）是否具有相同的值，应该使用方法equals，而不是==。

图3-3所示的程序对方法equals以及String中的方法equalsIgnoreCase的使用进行了说明。在这种表示法中，两个要进行相等性测试的对象是不对称的，因此开始看起来可能会有点儿别扭。例如：

```
s1.equals(s2)
s2.equals(s1)
```

这两个表达式是等价的。

```
import java.util.*;

public class StringEqualityDemo
{
    public static void main(String[] args)
    {
        String s1, s2;

        System.out.println("Enter two lines of text:");

        Scanner keyboard = new Scanner(System.in);
        s1 = keyboard.nextLine();
        s2 = keyboard.nextLine();

        if (s1.equals(s2))
            System.out.println("The two lines are equal.");
        else
            System.out.println("The two lines are not equal.");

        if (s2.equals(s1))
            System.out.println("The two lines are equal.");
        else
            System.out.println("The two lines are not equal.");

        if (s1.equalsIgnoreCase(s2))
            System.out.println(
                        "But the lines are equal, ignoring case.");
        else
            System.out.println(
                        "Lines are not equal, even ignoring case.");
    }
}
```

这两种对方法equals的调用是等价的。

屏幕对话示例

```
Enter two lines of text:
Java is not coffee.
Java is NOT COFFEE.
The two lines are not equal.
The two lines are not equal.
But the lines are equal, ignoring case.
```

图3-3 测试字符串是否相等

方法equalsIgnoreCase的表现与equals类似，只是在equalsIgnoreCase中，不区分字母的大小。例如，由于"Hello"和"hello"的第一个字符'H'和'h'是不同的字符，所以它们是不等的。但方

法equalsIgnoreCase会认为它们是相等的。例如，下列代码会输出Equal。

```
if ("Hello".equalsIgnoreCase("hello"))
    System.out.println("Equal");
```

注意，就像前面的equalsIgnoreCase那样，在String方法中使用引用字符串是完全合法的。引用字符串是String类型的对象，具有所有其他String类型对象所具有的方法。

对于本章中的各种应用程序来说，也可以用==来测试String类型对象的相等性，它会给出正确的结果。但在有些情况下，==无法正确地测试字符串是否相等，因此应该习惯于使用equals而不是==来测试字符串。△

记住：方法equals和equalsIgnoreCase

测试字符串是否相等时，不要使用==。作为替代，可以使用equals或equalsIgnoreCase。

语法：

String.equals(*Other_String*)
String.equalsIgnoreCase(*Other_String*)

举例（keyboard是一个Scanner对象）：

```
String s1;
s1 = keyboard.next();
if (s1.equals("Hello"))
    System.out.println("The string is Hello.");
else
    System.out.println("The string is not Hello.");
```

编程提示：字母顺序

程序经常要对两个字符串进行比较，以确定按字母顺序哪个字符串应该排在前面。Java没有内建的比较运算符可以用来比较字母顺序，但通过在图2-7中介绍过的两种String方法compareTo和toUpperCase，可以很方便地测试字符串的字母顺序。

方法compareTo会对两个字符串进行测试，以确定它们的字典排序。**字典排序**（lexicographic ordering）与字母排序类似，有时（但不总是）与字母排序相同。最简单的方式就是把字典排序当作字母排序，只是它具有不同的字母排列顺序。具体来说，在字典排序中，字母和其他字符是按附录C中列出的ASCII排序方式进行排序的。其中，所有的大写字母出现在所有的小写字母之前。例如，在字典顺序中，'Z'是在'a'之前出现的。因此，在对两个包含了大小写混合字母的字符串进行比较时，字典排序和字母排序是不同的。但是在附录C中所有的小写字母都是按照字母顺序排列的。因此，对任意两个全部由小写字母组成的字符串来说，字典排序和普通的字母排序是相同的。与之类似，在附录C的排序中，所有的大写字母也都是按照字母顺序排列的。因此，对任意两个全部由大写字母组成的字符串来说，字典排序和普通的字母排序也是相同的。因此，要按照（普通）字母顺序对两个由字母组成的字符串进行比较，只需要将两个字符串全部转换成大写字母（或全部转换成小写字母），然后对其进行字典排序就行了。下面看看Java中的一些细节。

如果s1和s2是两个已经赋了String值的String类型变量，那么，按照字典排序，若s1在s2之前，下列语句就会返回一个负数；如果两个字符串相同，就会返回零；如果S2在S1之前，就会返回一个正数。

```
s1.compareTo(s2)
```

因此，如果按照字典排序，s1在s2之前，那么，

```
s1.compareTo(s2) < 0
```

就会返回true（真），否则，就返回false（假）。例如，下列代码会产生正确的输出：

```
if (s1.compareTo(s2) < 0)
   System.out.println(
       s1 + " precedes " +  s2 + " in lexicographic ordering");
else if (s1.compareTo(s2) > 0)
   System.out.println(
       s1 + " follows " +  s2 + " in lexicographic ordering");
else //s1.compareTo(s2) == 0
   System.out.println(s1 + " equals " + s2);
```

正如前面提到的，一种用来测试两个字符串字母排序的方法是将它们全部转换成大写字母，然后用compareTo对s1和s2的大写版本进行字典排序测试。因此，下列代码也会产生正确的输出：

```
String upperS1 = s1.toUpperCase();
String upperS2 = s2.toUpperCase();
if (upperS1.compareTo(upperS2) < 0)
   System.out.println(
       s1 + " precedes " + s2 + " in ALPHABETIC ordering");
else if (upperS1.compareTo(upperS2) > 0)
   System.out.println(
       s1 + " follows " + s2 + " in ALPHABETIC ordering");
else //s1.compareTo(s2) == 0
   System.out.println(s1 + " equals " + s2 + " IGNORING CASE");
```

无论字符串s1和s2中包含了什么样的字符，前面的代码都可以编译并产生结果。但是，只有这两个字符串全部由字母组成时，字母排序才有意义，输出才有意义。

自测题

1. 假设goals是一个int类型变量。编写一条if-else语句，如果变量goals的值大于10，输出Wow，如果goals的值不超过10，输出Oh Well。
2. 假设goals和errors是int类型的变量。编写一条if-else语句，如果变量goals的值大于10，且errors的值为零，就输出单词Wow；否则，if-else语句输出Oh Well。
3. 假设salary和deductions是已经被赋值了的double类型变量。编写一条if-else语句，如果salary至少和deductions一样大，就输出OK，并将变量net设置为salary减去deductions。但是，如果salary小于deductions，if-else语句就只输出Crazy，而不改变任何变量的值。
4. 假设speed和visibility是int类型的变量。编写一条if-else语句，如果speed的值大于25，且visibility的值小于20，就将变量speed设置为25，并输出Caution。没有else部分。
5. 假设salary和bonus是double类型的变量。编写一条if-else语句，如果salary大于或等于MIN_SALARY，或者bonus大于或等于MIN_BONUS，就输出OK。否则，输出Too low。MIN_SALARY和MIN_BONUS是命名常量。
6. 假设nextWord是一个String变量，已经为其赋了一个全部由字母组成的String值。编写一些Java代码，如果按照字母排序nextWord在"N"之前，就输出消息"First half of the alphabet"。如果按照字母排序nextWord不在"N"之前，输出"Second half of the alphabet"。（注意，"N"使用了双引号，表示它是一个String值，而不是用单引号表示的char值。）
7. 假设x1和x2是两个已经赋了值的变量。这两个变量都为int类型时，应该如何测试它们是否相等？这两个变量都为String类型时，应该如何测试它们是否相等？

3.1.3 嵌套语句与复合语句

注意，一条if-else语句中也可包含两条较小的语句。例如，考虑下列语句：

```
if (balance >= 0)
    balance = balance + (INTEREST_RATE * balance)/12;
else
    balance = balance - OVERDRAWN_PENALTY;
```

这条语句中包含了下列两条较小的语句：

```
balance = balance + (INTEREST_RATE * balance)/12;
balance = balance - OVERDRAWN_PENALTY;
```

注意，这些较小的语句要比if和else多缩进一级。

if-else语句中可以包含任何类型的语句。尤其是可以在一条if-else语句中使用另一条if-else语句，如下所示：

```
if (balance >= 0)
    if (INTEREST_RATE >= 0)
        balance = balance + (INTEREST_RATE * balance)/12;
    else
        System.out.println("Cannot have a negative interest. ");
else
    balance = balance - OVERDRAWN_PENALTY;
```

如果balance的值大于或等于零，就执行下面整个if-else语句：

```
if (INTEREST_RATE >= 0)
    balance = balance + (INTEREST_RATE * balance)/12;
else
    System.out.println("Cannot have a negative interest.");
```

稍后将介绍在一条if-else语句中嵌套if-else语句的最常见方式。

另一种简单实用的、将较小语句嵌套到较大语句中的方法是：将一串语句放在花括号{}中。将一串语句包含在花括号中时，这些语句会被当成一个较大的语句。因此，下面就是一个包含了两条较小语句的较大语句：

```
{
    System.out.println("Good for you. You earned interest.");
    balance = balance + (INTEREST_RATE * balance)/12;
}
```

这些由花括号将一串语句包围起来形成的语句称为**复合语句**（compound statement）。它们很少单独使用，但经常作为if-else这样较大语句的子语句使用。前面的复合语句可能会出现在下面这样的if-else语句中：

```
if (balance >= 0)
{
    System.out.println("Good for you. You earned interest.");
    balance = balance + (INTEREST_RATE * balance)/12;
}
else
{
    System.out.println("You will be charged a penalty.");
    balance = balance - OVERDRAWN_PENALTY;
}
```

注意，复合语句可以简化对if-else语句的描述。一旦了解了复合语句，可以说每条if-else语句都是如下形式的：

```
if (Boolean_Expression)
```

```
        Statement_1
    else
        Statement_2
```

如果希望一个分支中包含多条而不是一条语句，就可以使用复合语句。从技术上来讲，复合语句就是一条语句，因此，从技术上来讲，前面（以`if (balance >= 0)`开头）的`if-else`语句的每条分支都是一条单语句。

Java提示：匹配else和if

在编写嵌套的if-else语句时，有时可能会分不清哪个if与哪个else配套的。要消除这种混乱，可以像使用圆括号一样，用花括号将语句组织起来。

例如，下面是在本章前面使用过的嵌套语句：

```
if (balance >= 0)
   if (INTEREST_RATE >= 0)
       balance = balance + (INTEREST_RATE * balance)/12;
   else
       System.out.println("Cannot have a negative interest. ");
else
   balance = balance - OVERDRAWN_PENALTY;
```

使用一些附加的花括号可以使上述语句更清晰一些，如下所示。

```
if (balance >= 0)
{
   if (INTEREST_RATE >= 0)
       balance = balance + (INTEREST_RATE * balance)/12;
   else
       System.out.println("Cannot have a negative interest. ");
}
else
   balance = balance - OVERDRAWN_PENALTY;
```

在这种情况下，花括号对程序的清晰度有所帮助，但严格来讲，这些花括号并不是必须的。在其他一些情况下，它们则是必须的。如果省略掉一个else，事情就会变得有点儿棘手了。下面两条语句唯一的区别就是有一条语句中包含了一对花括号，但它们却具有不同的含义。

```
//First Version
if (balance >= 0)
{
   if (INTEREST_RATE >= 0)
       balance = balance + (INTEREST_RATE * balance)/12;
}
else
   balance = balance - OVERDRAWN_PENALTY;
//second version
if (balance >= 0)
   if (INTEREST_RATE >= 0)
       balance = balance + (INTEREST_RATE * balance)/12;
else
   balance = balance - OVERDRAWN_PENALTY;
```

在一条if-else语句中，每个else都是与最近的未匹配if配成一对的。因此，在第二个（没有花括号的）版本中，else是和第二个if配成一对的，因此，第二个版本的含义为：

```
//Equivalent to Second Version
if (balance >= 0)
{
```

```
    if (INTEREST_RATE >= 0)
        balance = balance + (INTEREST_RATE * balance)/12;
    else
        balance = balance - OVERDRAWN_PENALTY;
}
```

为了使两者的区别更清晰一些，考虑一下当balance大于或等于0时，会发生什么情况。在第一个版本中，这会引发下列动作：

```
if (INTEREST_RATE >= 0)
    balance = balance + (INTEREST_RATE * balance)/12;
```

在第一个版本中，如果balance不大于或等于0，就会执行下列动作：

```
balance = balance - OVERDRAWN_PENALTY;
```

在第二个版本中，如果balance大于或等于0，会执行下面整个if-else语句：

```
if (INTEREST_RATE >= 0)
   balance = balance + (INTEREST_RATE * balance)/12;
else
   balance = balance - OVERDRAWN_PENALTY;
```

在第二个版本中，如果balance不大于或等于0，则不会执行任何动作。

3.1.4 多分支if-else语句

如果能够向两个方向分支，就能够向4个方向分支。可以只向两个方向分支，然后让这两个分支中的每一个都向两个方向分支。通过这种技巧，就可以用嵌套式if-else语句产生能够分出任意数量分支的多路分支。有一种标准的方法可以做到这一点。实际上，这种方法非常标准，以至于会被当成一种新的分支语句来对待，而不只是一种由很多嵌套的if-else语句组成的嵌套语句。下面来看一个例子。

假设balance是一个用来设置支票账户余额的变量，而你想知道自己的余额到底是正、是负（透支）、还是零（为了避免任何与精确性有关的问题，假设balance是int类型的。具体来说，把balance当成账户中忽略了美分的美元数）。要弄清你的账户是正、是负还是零，可以使用下列嵌套式if-else语句：

```
if (balance > 0)
    System.out.println("Positive balance");
else if (balance < 0)
    System.out.println("Negative balance");
else if (balance == 0)
    System.out.println("Zero balance");
```

首先，要注意这条语句的缩进方式。这是对多分支if-else语句进行缩进的常见方式。一条多分支if-else语句实际上就是一个普通的嵌套式if-else语句，但是，我们对它的缩进方式反映了对多分支if-else语句的认识方法。

执行一条多分支if-else语句时，计算机会从最上面开始逐个地对布尔表达式进行测试。找到第一个为真的布尔表达式时，就执行跟在结果为真的布尔表达式后面的语句。例如，如果balance大于零，前面的代码就会输出"Positive balance"；如果balance小于0，就会输出"Negative balance"；如果balance等于0，就会输出"Zero balance"。根据变量balance的值，只会生成3种可能输出中的一种。

在这个例子中，有3种可能性，但可以提供任意数量的可能性；只要添加更多的else-if部分就行了。

在这个例子中，各种可能性是互斥的。但你可以使用任意的布尔表达式，即使它们不是互斥的也可以。如果有多个布尔表达式为真，它只会执行与第一个为真的布尔表达式相关的动作。多分支if-else语句决不会执行多个动作。

如果布尔表达式都不为真，则什么事情都不会发生。但是，最好在末尾处添加一条不含任何if的else子句，这样当布尔表达式都不为真时，就会执行else子句。实际上，可以用这种方法重新编写原来那个关于支票账户余额的例子。很显然，如果balance不为正，也不为负，那它一定为零。因此我们不需要对if (balance == 0)进行测试。前面的多分支if-else语句等同于下列语句：

```
if (balance > 0)
    System.out.println("Positive balance");
else if (balance < 0)
    System.out.println("Negative balance");
else
    System.out.println("Zero balance");
```

编程示例：进行字母分级

图3-4中包含了一个程序，这个程序是根据传统的规则来进行字母分级的，即90或90分以上为A，80或80分以上（到90为止）为B，依此类推。

```
import java.util.*;

public class Grader
{
    public static void main(String[] args)
    {
        int score;
        char grade;

        System.out.println("Enter your score: ");
        Scanner keyboard = new Scanner(System.in);
        score = keyboard.nextInt();

        if (score >= 90)
            grade = 'A';
        else if (score >= 80)
            grade = 'B';
        else if (score >= 70)
            grade = 'C';
        else if (score >= 60)
            grade = 'D';
        else
            grade = 'F';

        System.out.println("Score = " + score);
        System.out.println("Grade = " + grade);
    }
}
```

屏幕对话示例

```
Enter your score:
85
Score = 85
Grade = B
```

图3-4 多分支if-else语句

快速参考：多分支if-else语句

语法：

```
if (Boolean_Expression_1)
    Action_1
else if (Boolean_Expression_2)
    Action_2
    ⋮
else if (Boolean_Expression_n)
    Action_n
else
    Default_Action
```

举例：

```
if (number < 10)
    System.out.println("number < 10");
else if (number < 50)
    System.out.println("number >= 10 and number < 50");
else if(number < 100)
    System.out.println("number >= 50 and number < 100");
else
    System.out.println("number >= 100.");
```

这些动作都是Java语句。对这些布尔表达式的测试是从最上面一个开始，逐个地进行的。找到第一个为真的布尔表达式时，执行跟在结果为真的布尔表达式后面的动作。如果所有布尔表达式为真，就执行*Default_Action*。

要注意，任何多分支if-else语句都是按顺序对布尔表达式进行检测的，因此，除非发现第一个布尔表达式为假，否则不会对第二个布尔表达式进行检测。因此，在检测第二个布尔表达式时，我们就知道第一个布尔表达式是假的了，这样，如果检测到了第二个布尔表达式，score < 90就是成立的。因此，如果用

```
((score >= 80) && (score < 90))
```

来取代

```
(score >= 80)
```

多分支if-else语句会具有同样的含义。

对每个布尔表达式使用同样的推理方法，就会发现图3-4中的多分支if-else语句等价于下列语句：

```
if (score >= 90)
    grade = 'A';
else if ((score >= 80) && (score < 90))
    grade = 'B';
else if ((score >= 70) && (score < 80))
    grade = 'C';
else if ((score >= 60) && (score < 70))
    grade = 'D';
else
    grade = 'F';
```

大多数程序员都会使用图3-4所示的版本，因为那种版本更有效也更优美一些，但这两种版本都是可以的。

自测题

8. 下列代码会产生什么样的输出？

```
int time = 2, tide = 3;
if (time + tide > 6)
    System.out.println("Time and tide wait for no one.");
else
    System.out.println("Time and tide wait for me.");
```

9. 下列代码会产生什么样的输出?

```
int time = 4, tide = 3;
if (time + tide > 6)
    System.out.println("Time and tide wait for no one.");
else
    System.out.println("Time and tide wait for me.");
```

10. 下列代码会产生什么样的输出?

```
int time = 2, tide = 3;
if (time + tide > 6)
    System.out.println("Time and tide wait for no one.");
else if (time + tide > 5)
    System.out.println("Time and tide wait for some one.");
else if (time + tide > 4)
    System.out.println("Time and tide wait for everyone one.");
else
    System.out.println("Time and tide wait for me.");
```

11. 假设number是一个已经赋了值的int类型变量。编写一条多分支if-else语句，如果number大于10，输出High，如果number小于5，输出Low，如果number是其他任何值，输出So-so。

3.1.5 switch语句

switch语句是多路分支语句，它会根据一个整数或字符表达式的值来决定进入哪条分支。图3-5显示了一条示例switch语句。switch语句以一个关键字switch开头，后面跟着一个包含在圆括号中的表达式。在图3-5中，表达式就是变量numberOfBabies。这个表达式被称为**控制表达式**（controlling expression）。

```
import java.util.*;

public class MultipleBirths
{
    public static void main(String[] args)
    {
        int numberOfBabies;
        System.out.print("Enter number of babies: ");
        Scanner keyboard = new Scanner(System.in);
        numberOfBabies = keyboard.nextInt( );

        switch (numberOfBabies)
        {
            case 1:
                System.out.println("Congratulations.");
                break;
            case 2:
                System.out.println("Wow. Twins.");
                break;
            case 3:
```


图3-5 switch语句

```
            System.out.println("Wow. Triplets.");
            break;
        case 4:
        case 5:
            System.out.println("Unbelieveable.");
            System.out.println(numberOfBabies + " babies");
            break;
        default:
            System.out.println("I don' t believe you.");
            break;
    }
  }
}
```

屏幕对话示例1

```
Enter number of babies:1
Congratulations.
```

屏幕对话示例2

```
Enter number of babies:3
Wow.Triplets.
```

屏幕对话示例3

```
Enter number of babies:4
Unbelievable.
4 babies
```

屏幕对话示例4

```
Enter number of babies:6
I don't believe you.
```

图3-5 (续)

这个表达式的下面是一系列包含在花括号中的case，每个case都由关键字case及其后面的一个常量、一个冒号以及一系列语句组成，这一系列语句就是case要执行的动作。放在case后面的常量被称为**case标记**（case label)。执行switch语句时，将控制表达式——在这个例子中就是numberOfBabies——计算出来。然后，对选项列表进行搜索，直到找到一个与控制表达式匹配的case标记为止，执行与那个标记相关的动作。不能使用重复的case标记，那样会造成一种模棱两可的局面。如果没有找到匹配的标记，就执行标记为default的分支。

default分支是可选的。如果没有default分支，又找不到任何与之匹配的case语句，就不会执行任何动作。尽管default分支是可选的，但最好还是使用它。如果你认为在没有default分支的情况下，列出的case覆盖了所有可能的情况，可以插入一条错误消息作为default分支。这样就能及时发现你是不是错过了一些模糊的case分支。

注意，图3-5中的每个case分支的动作都是以单词break作为结束的。这是一条break语句，用来终止switch语句。break语句由单词break及其后跟着的分号组成。如果没有break语句，就会继续执行下一个case分支中的动作，直到碰到一条break语句，或抵达了switch语句的末尾为止。

有时你可能需要一个没有break语句的case分支。在一个case分支中不能有多个标记，但可以逐个列举case分支，使它们都适用于相同的动作。例如，在图3-5中，由于case 4分支中没有break语句（实际上，case 4分支中根本没有动作语句），所以case 4分支和case 5分支会产生相同的分支动作。

下面看一个switch语句：

```
switch (eggGrade)
{
    case 'A':
    case 'a':
        System.out.println("Grade A");
        break;
    case 'C':
    case 'c':
        System.out.println("Grade C");
        break;
    default:
        System.out.println("We only buy grade A and grade C.");
        break;
}
```

在这个例子中，变量eggGrade应该是char类型。

注意，在这些case分支中，并不需要形成任何类型的范围，就像在前面的例子中那样，可以有'A'和'C'而没有'B'。类似地，在一个使用整数case标记的switch语句中，可以有整数1和3，而没有2。

switch语句中的控制表达式不一定要是单变量。它可以是一个包含+、*或其他运算符的复杂表达式，但表达式经计算得出的必须是int这样的整数类型的值或char类型的值。

名为“switch语句”的快速参考对swtich语句的语法进行了解释。一定要注意case标记后面的冒号。

快速参考：switch语句

语法：

```
switch (Controlling_Expression)
{
    case Case_Label:
        Statement;
        Statement;
        ...
        Statement;
        break;
    case Case_Label:
        Statement;
        Statement;
        ...
        Statement;
        break;
```

每个*Case_Label*都是与*Controlling_Expression*类型相同的常量。每个case分支的*Case_Label*都应该是不同的。*Controlling_Expression*必须是char、int、short或byte类型的。

break语句可以省略。如果没有break语句，就继续执行下一个case分支。

<上面这样的case分支可以有任意个。下面的default case分支是可选的。>

```
    default:
        Statement;
```

```
        Statement;
        ...
        Statement;
        break;
}
```

举例：

```
int seatLocationCode;
...
switch (seatLocationCode)
{
    case 1:
        System.out.println("Orchestra.");
        price = 40.00;
        break;
    case 2:
        System.out.println("Mezzanine.");
        price = 30.00;
        break;
    case 3:
        System.out.println("Balcony.");
        price = 15.00;
        break;
    default:
        System.out.println("Unknown ticket code.");
        break;
}
```

▲ 易犯错误：遗漏了一条break语句

对一个包含switch语句的程序进行测试，如果在你期望它只执行一个case分支时，它执行了两个case分支，那就很可能是你忘记在需要的地方加break语句。 △

3.1.6 条件运算符（选读）

为了与较早的编程风格保持兼容，Java中包含了一种运算符，这种运算符是对某些形式的if-else语句表示法的简化。例如，考虑下列语句：

```
if (n1 > n2)
    max = n1;
else
    max = n2;
```

可以用条件运算符将上述语句表示成如下形式：

```
max = (n1 > n2) ? n1 : n2;
```

赋值语句右边的表达式(n1 > n2) ? n1 : n2就是条件运算符表达式。?和:一起被称为**条件运算符**（conditional operator）或**三元运算符**（ternary operator）。**条件运算符表达式**（conditional operator expression）以一个布尔表达式开头，后面跟着一个?，以及两个由冒号分隔的表达式。如果布尔表达式为真，就返回第一个表达式；否则，就返回第二个表达式。

如这个例子所示，条件运算符最常见的用途是，根据布尔条件，将两个不同的值中的一

个赋给一个变量。一定要注意，条件表达式通常会返回一个值，因此，它只是等效于某种特定类型的if-else语句。另一个例子可能有助于说明条件运算符的用法。

```
if (hoursWorked<= 40)
    pay = hoursWorked * payRate;
else
    pay = 40 * payRate + 1.5 * (hoursWorked - 40) * payRate;
```

代码说明雇员的工资等于单位工资乘以工作的小时数，但是，如果雇员的工作超过了40小时，超过40小时的那部分时间都要按照正常单位工资的1.5倍来付款。可以通过条件运算符以下列形式表示这个过程：

```
pay =
    (hoursWorked <= 40) ?
        (hoursWorked * payRate) :
        (40 * payRate + 1.5 * (hoursWorked - 40) * payRate);
```

自测题

12. 下列代码会产生什么样的输出？

```
int code = 2;
switch (code)
{
    case 1:
        System.out.println("Hello.");
    case 3:
        System.out.println("Good-bye.");
        break;
    default:
        System.out.println("Till we meet again.");
        break;
}
```

13. 假设你修改了第12题中的代码，使第一行如下：

```
int code = 1;
```

会产生什么样的输出？

14. 下列代码会产生什么样的输出？

```
char letter = 'B';
switch (letter)
{
    case 'A':
    case 'a':
        System.out.println("Some kind of A.");
    case 'B':
    case 'b':
        System.out.println("Some kind of B.");
        break;
    default:
        System.out.println("Something else.");
        break;
}
```

15. 下列代码会产生什么样的输出？

```
int key = 1;
switch (key + 1)
{
```

```
    case 1:
        System.out.println("Cake");
        break;
    case 2:
        System.out.println("Pie");
        break;
    case 3:
        System.out.println("Ice cream");
    case 4:
        System.out.println("Cookies");
        break;
    default:
        System.out.println("Diet time");
}
```

16. 假设你对第15题中的代码进行修改，使第一行如下：

```
int key = 3;
```

会产生什么样的输出？

17. 假设对第15题中的代码进行修改，使第一行如下：

```
int key = 5;
```

会产生什么样的输出？

3.2 Java循环语句

再来一次。

——贝西伯爵乐队，唱片“巴黎的四月”

“再来一遍，山姆。”

——被（误）认为出现在电影《卡萨布兰卡》中，
《卡萨布兰卡》中确实有诸如“来一遍，山姆。”或与之类似的话

程序经常需要重复某些动作。例如，一个分级程序中会包含一些代码，根据学生作业和考试的分数，为学生指定一个等级。要为全班学生指定等级，程序就要为班上的每个学生重复这个动作。程序中重复一条或一组语句的部分被称作**循环**（loop）。循环中重复的语句（或语句组）被称为循环的**主体**（body）。循环主体的每次重复被称为一次循环**迭代**（iteration）。

设计一个循环时，要确定循环主体需要执行什么动作，还要确定一种机制，用来判定循环应该在什么时候停止执行循环体。

3.2.1 while语句

在Java中构建循环的一种方式是使用**while语句**，也称为**while循环**。while语句会一次次地重复它的动作，直到控制布尔表达式为假为止。这就是它被称为while循环的原因：当（while）控制布尔表达式为真时，就重复循环过程。例如，图3-6中包含了一个小型的while语句实例。语句以关键字while开始，后面跟着一个用圆括号括起来的布尔表达式——控制布尔表达式。当控制布尔表达式为真时，就重复循环体（被重复的那部分语句）。循环体是一条语句，通常是一条用花括号{}括起来的复合语句。循环体中通常会包含一些可以将控制布尔表达式的值由真改为假的动作，以终止循环。下面单步执行一下图3-6中的while循环。

考虑一下图3-6中while语句的第一个对话示例中，用户输入2，这个2就成了变量number的值。控制布尔表达式如下：

```
(count <= number)
```

因为count为1，number为2，这个布尔表达式为真，因此，会执行如下所示的循环体：

```
{
    System.out.println(count + ",  ");
    count++;
}
```

```
import java.util.*;

public class WhileDemo
{
    public static void main(String[] args)
    {
        int count, number;

        System.out.println("Enter a number");
        Scanner keyboard = new Scanner(System.in);
        number = keyboard.nextInt();

        count = 1;
        while (count <= number)
        {
            System.out.print(count + ", ");
            count++;
        }
        System.out.println();
        System.out.println("Buckle my shoe.");
    }
}
```

屏幕对话示例1

```
Enter a number:
2
1, 2,
Buckle my shoe.
```

屏幕对话示例2

```
Enter a number:
3
1, 2, 3
Buckle my shoe.
```

屏幕对话示例3

```
Enter a number:
0

Buckle my shoe.
```

循环体被迭代了零次。

图3-6 while循环

循环体将count的值写到屏幕上，然后将count的值加1，因此，1被写到屏幕上，而count的值变成了2。

循环体迭代一次之后，会再次对控制布尔表达式进行检查。因为count为2，number也为2，所以布尔表达式仍然为真。因此会再执行一次循环体，再次将count的值写到屏幕上，并再次将count的值加1，因此，2被写到屏幕上去，而count的值变成3了。

循环体迭代两次之后，会再次检查控制布尔表达式。现在count的值为3，而number的值还是2，因此，下面显示的控制布尔表达式现在就为假了：

```
(count <= number)
```

因为控制布尔表达式的值为假，while循环终止，程序会继续执行while语句之后的两条System.out.println语句。第一条System.out.println语句在while循环输出的数字行后加回车符，第二条语句输出了"Buckle my shoe."。

所有while语句的构建方式都与图3-6显示的示例类似。语句的一般形式为：

```
while(Boolean_Expression)
    Body_Statement
```

*Body_Statement*可以是一条简单的语句，例如：

```
while (next > 0)
    next = keyboard.nextInt();
```

但更常见的情况与图3-6类似，*Body_Statement*是一条复合语句，因此while循环最常见的形式如下：

```
while(Boolean_Expression)
{
    First_Statement
    Second_Statement
    ...
    Last_Statement
}
```

图3-7对while循环的语义（含义）进行了描述。

while (*Boolean_Expression Bady*)

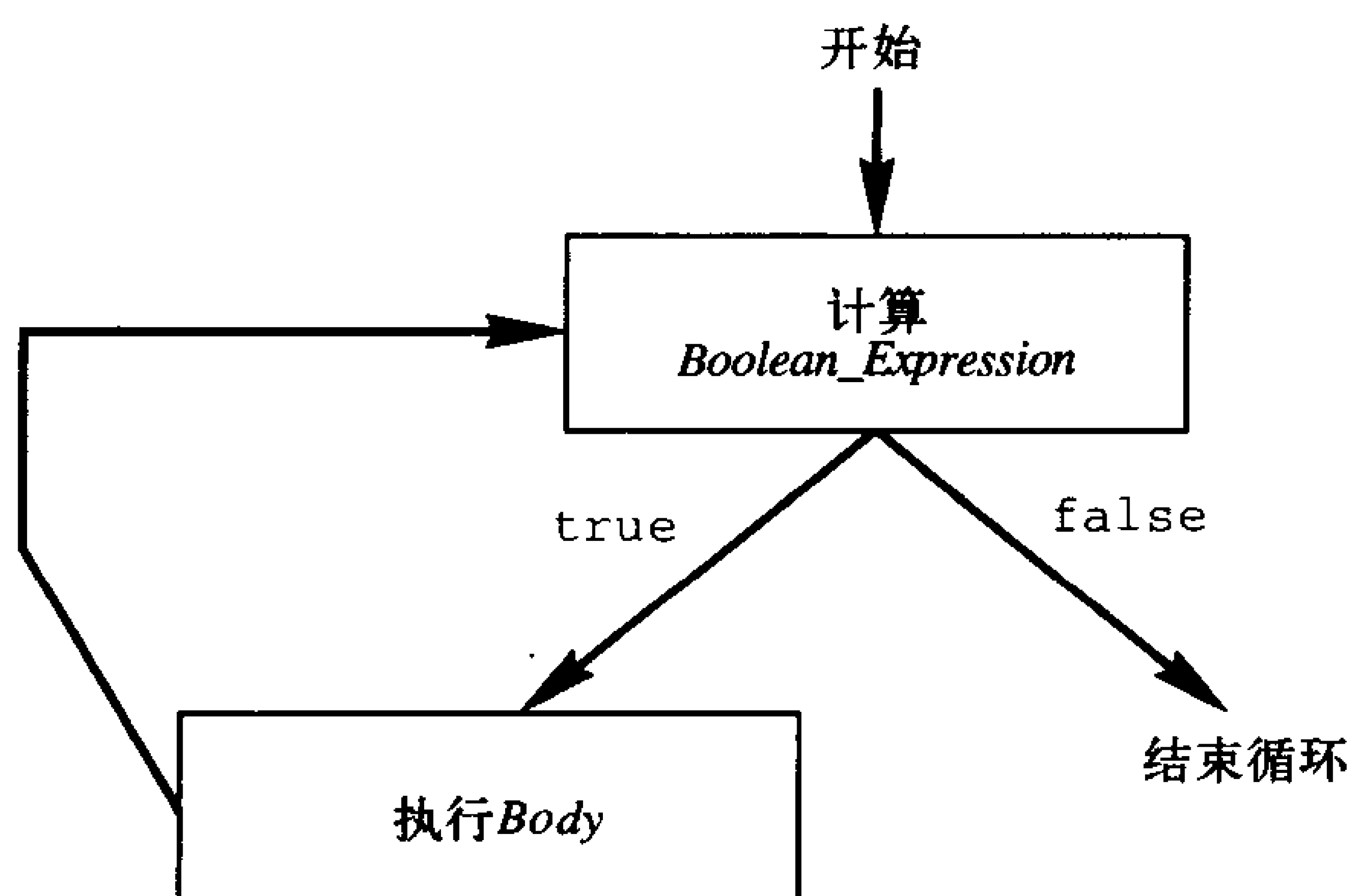


图3-7　while语句的语义

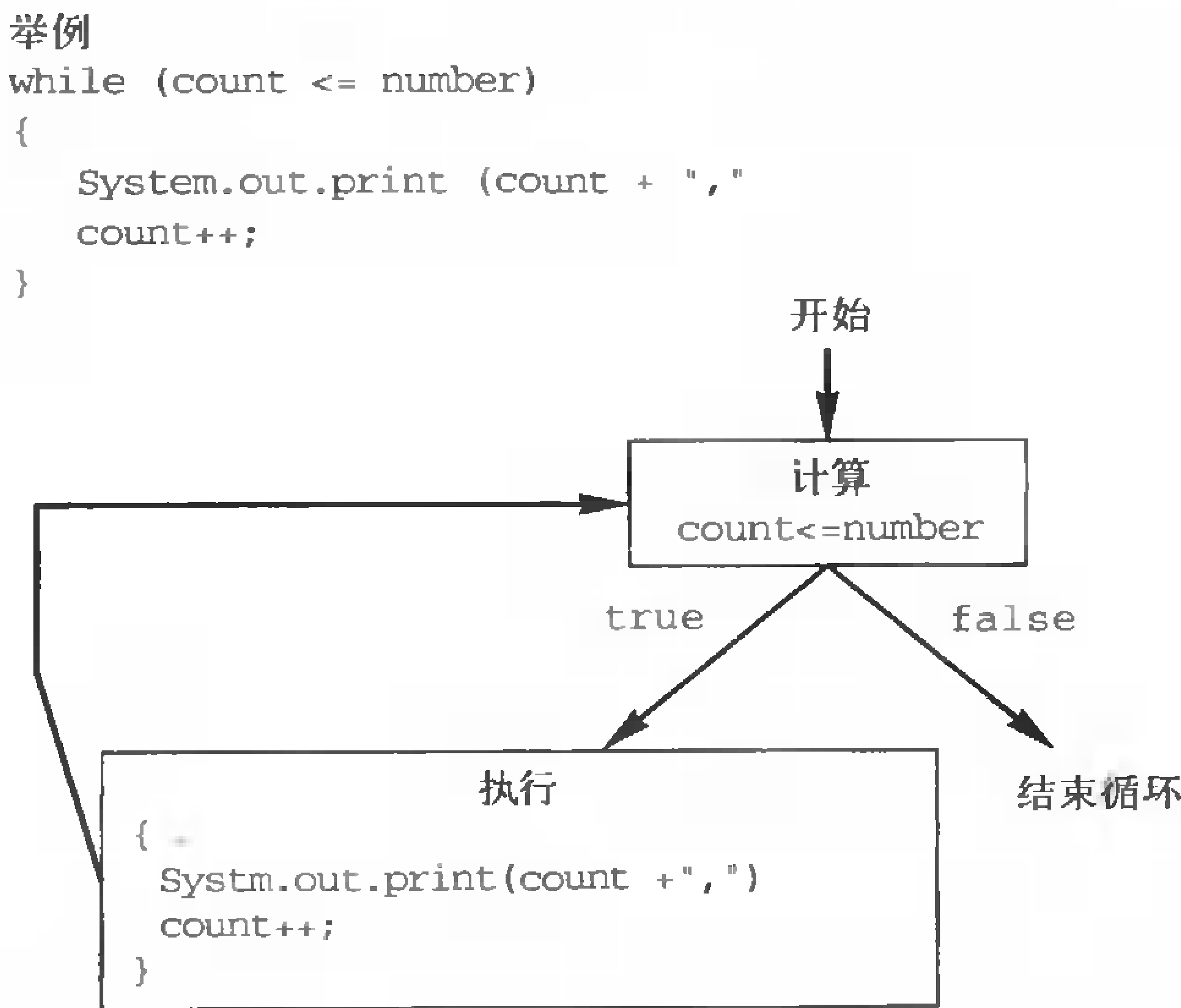


图3-7 （续）①

快速参考：while语句

语法：

while(*Boolean_Expression*)
    Body

*Body*可能是一条单语句，更常见的是由包含在花括号{}中的由一串语句组成的复合语句。

举例（keyboard是一种常见的Scanner对象）：

```
while (next > 0)
{
    next = keyboard.nextInt();
    total = total + next;
}
```

■ Java提示：while循环可以执行零次迭代

while循环的主体可以执行零次。执行while循环时发生的第一件事就是检查控制布尔表达式。如果控制布尔表达式为假，就不执行循环体，一次也不执行。这看起来可能有些奇怪。说到底，如果不执行循环体，为什么要编写这个循环呢？答案是：你可能希望一个循环能够根据用户的输入来决定将循环体执行零次或多次。可能循环是要将你一天所有账单的总数加出来。如果没有账单，根本就不希望执行循环体。图3-6中的“屏幕对话示例3”显示的就是一个while循环的小型实例，这个实例的循环体迭代了零次

3.2.2 do-while语句

do-while语句（也称为**do-while循环**）与while语句很类似。它们主要的区别在于，使用do-while语句时，循环体至少会执行一次，而while循环的主体可能会执行零次。图3-8中

① 稍后会讨论在循环内部break语句的用法。这里的语义假定在循环体中没有break语句。

包含了一个与图3-6中的while循环类似（但不相同）的do-while循环示例。注意，采用do-while循环时，即如“屏幕对话示例3”所示，即便布尔表达式一开始就是假的，循环体也至少会执行一次。

```
import java.util.*;

public class DoWhileDemo
{
    public static void main("String[] args")
    {
        int count, number;

        System.out.println("Enter a number");
        Scanner keyboard = new Scanner(System.in);
        number = keyboard.nextInt();

        count = 1;
        do
        {
            System.out.print(count + ", ");
            count++;
        }while (count <= number);

        System.out.println();
        System.out.println("Buckle my shoe.");
    }
}
```

屏幕对话示例1

```
Enter a number:
2
1, 2,
Buckle my shoe.
```

屏幕对话示例2

```
Enter a number:
3
1, 2, 3,
Buckle my shoe.
```

屏幕对话示例3

```
Enter a number:
0
1,
Buckle my shoe.
```

循环体至少要执行一次。

图3-8　do-while循环

do-while语句的语法如下所示：

```
do
    Body_Statement
while(Boolean_Expression);
```

*Body_Statement*可以是一条单语句，如下例所示：

```
do
    next = keyboard.nextInt();
while (next > 0);
```

但在更多的情况下是如图3-8所示，*Body_Statement*会是一条复合语句，因此do-while循环最常见的形式为

```
do
{
    First_Statement
    Second_Statement
      ...
    Last_Statement
}While (Boolean_Expression);
```

一定要注意圆括号中*Boolean_Expression*后面的分号。

注意，我们将末端的花括号}和while语句放在了同一行上。有些程序员喜欢将它们放在不同的行上。任何一种形式都可以，但要保持一致。

执行do-while循环时，发生的第一件事是执行循环体。之后，do-while循环的行为就和while循环完全一样了。它会检查布尔表达式。如果布尔表达式为真，就再次执行循环体；如果布尔表达式为假，循环就结束了。只要布尔表达式为真，就一次接一次地执行这个过程。

尽管我们不推荐用这种方式重写do-while循环，但看看下面进行的重写，可能会有助于理解do-while循环。可以将图3-8的do-while循环写成下列包含了一个while循环的等效代码：

```
{
    System.out.print (count + ", ");
    count++;
}
while(count <= number)
{
    System.out.print (count + ", ");
    count++;
}
```

以这种方式来看时，很明显，do-while循环和while循环只在一个细节上有所不同。使用do-while循环时，循环体总是至少会执行一次；使用while循环时，循环体可能会执行零次。

图3-9显示了do-while循环的语义（含义）。

快速参考：do-while语句

使用do-while语句，循环体至少会执行一次。

语法：

```
do
    Body
while(Boolean_Expression);
```

*Body*可能是一条单语句，更常见的是由包含在花括号{}中的一系列语句组成的复合语句。一定要注意圆括号中*Boolean_Expression*后面的分号。

举例：

```
do
{
    next = keyboard.nextInt();
    total = total + next;
}while(next > 0);
```

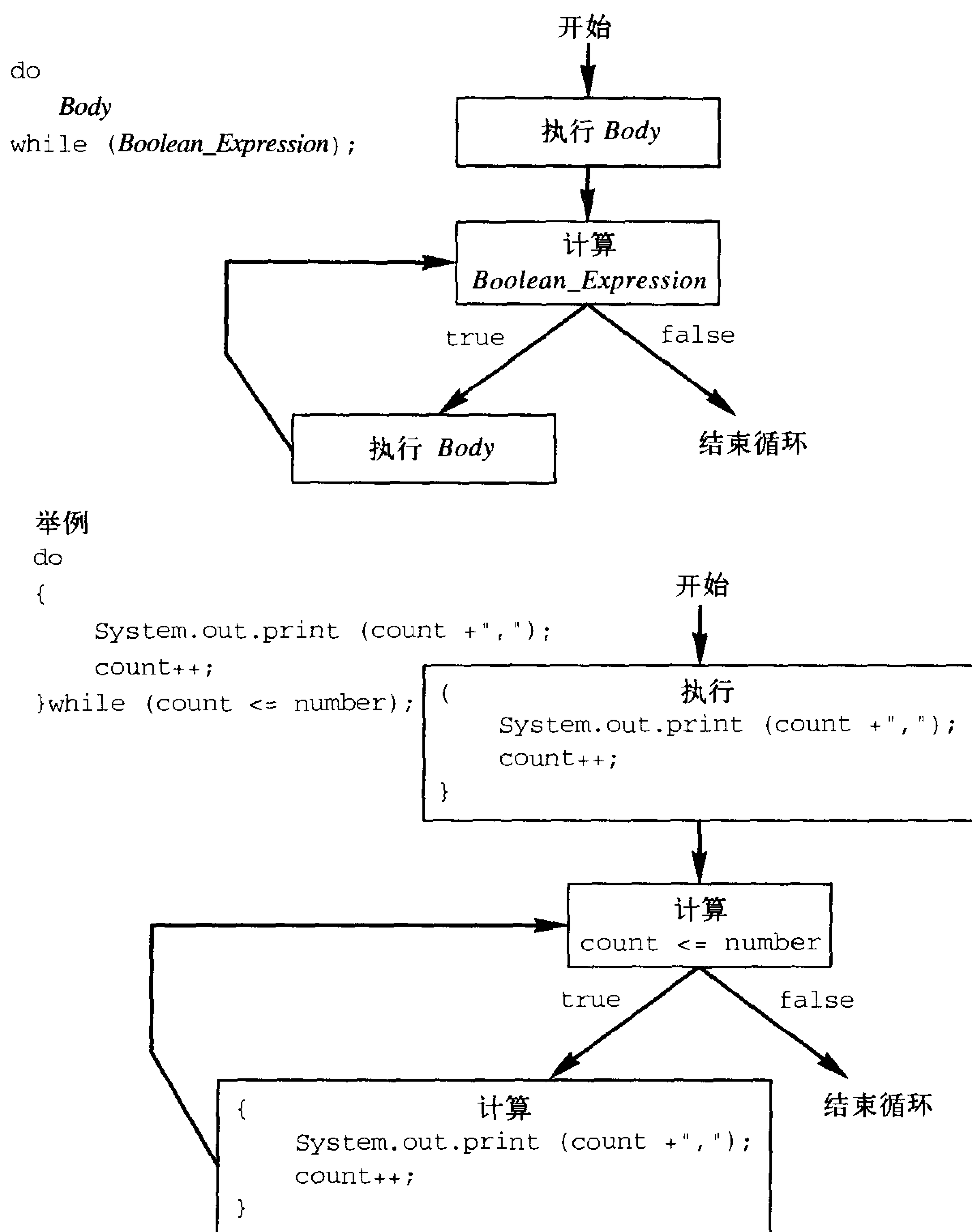


图3-9　do-while语句的语义①

编程示例：bug侵扰

家乡遭到了蟑螂的侵扰。这不是个让人高兴的话题，但幸运的是，有一家名为除虫专家的本地公司有一种可以将蟑螂从房子里面清除出去的方法。就像那句格言所说的："这是件肮脏的工作，但总得有人来做。"唯一的问题是镇上的居民们太自满了，在情况变得无法控制之前不会去根除蟑螂。因此，公司在本地的大型购物中心安装了一台计算机，以便让人们了解房间里的问题会有多么严重。图3-10所示为运行在这台计算机上的程序。

已定义常量给出了与这类蟑螂有关的基本情况。对蟑螂来说，它们的数量增长是相对比较缓慢的，但情况仍然非常严重。未经验证的消息表明，蟑螂的数量基本上每周要翻一倍。如果数量每周翻一倍，增长率将是每周100%，但幸运的是，这类蟑螂的每周增长率只有95%。这些蟑螂还非常大。如果用立方英尺来表示，它们的平均大小为0.002立方英尺（只是比0.3立方英寸稍小一点）。程序中确实进行了一些简化计算的假设。程序假设房间里没有家具，并假设蟑螂填满房间时中间不会有空隙。实际情况可能会比这个使用了简化假设的程序所描述的情况更加严重。

① 稍后会讨论在循环内部break语句的用法。这里的语义假定在循环体中没有break语句。

```
import java.util.*;

/**
 Program to calculate how long it will take a population of
 roaches to completely fill a house from floor to ceiling.
*/
public class BugTrouble
{
    public static final double GROWTH_RATE = 0.95;//95% per week
    public static final double ONE_BUG_VOLUME = 0.002;//cubic feet

    public static void main(String[] args)
    {
        System.out.println("Enter the total volume of your house");
        System.out.print("in cubic feet: ");
        Scanner keyboard = new Scanner(System.in);
        double houseVolume = keyboard.nextDouble();

        System.out.println("Enter the estimated number of");
        System.out.print("roaches in your house: ");
        int startPopulation = keyboard.nextInt();
        int countWeeks = 0;
        double population = startPopulation;
        double totalBugVolume = population*ONE_BUG_VOLUME;

        while (totalBugVolume < houseVolume)
        {
            population = population + (GROWTH_RATE*population);
            totalBugVolume = population*ONE_BUG_VOLUME;
            countWeeks++;
        }

        System.out.println("Starting with a roach population of "
                                      + startPopulation);
        System.out.println("and a house with a volume of "
                                + houseVolume + " cubic feet,");
        System.out.println("after " + countWeeks + " weeks,");
        System.out.println("the house will be filled");
        System.out.println("floor to ceiling with roaches.");
        System.out.println("There will be " + (int)population + " roaches.");
        System.out.println("They will fill a volume of "
                          + (int)totalBugVolume + " cubic feet.");

        System.out.println("Better call Debugging Experts Inc.");
    }
}
```

正如第2章中介绍的，(int)是一种强制类型转换。

屏幕对话示例

```
Enter the total volume of your house
in cubic feet: 20000
Enter the estimated number of
roaches in your house: 100
Starting with a roach population of 100
and a house with a volume of 20000.0 cubic feet,
after 18 weeks,
the house will be filled
floor to ceiling with roaches.
There will be 16619693 roaches.
They will fill a volume of 33239 cubic feet.
Better call Debugging Experts Inc.
```

图3-10　计算蟑螂数量的程序

下面来看一下图3-10程序中的while循环：

```
while( totalBugVolume < houseVolume)
{
    population = population + (GROWTH_RATE * population);
    totalBugVolume = population * ONE_BUG_VOLUME;
    countWeeks++;
}
```

这个循环只是用下列语句更新了蟑螂的数量和蟑螂的体积，这些语句说明了在一周内蟑螂的数量和体积是如何变化的：

```
population = population + (GROWTH_RATE * population);
totalBugVolume = population * ONE_BUG_VOLUME;
```

由于增长率和一只蟑螂的体积都是正的，所以，population的值会随着每次循环迭代而增加，因此totalBugVolume的值也会随着每次循环迭代而增加。所以，totalBugVolume的值最终会超过houseVolume的值，复制在这里的控制布尔表达式也会变成假的，从而结束while循环：

```
(totalBugVolume < houseVolume)
```

变量countWeeks从零开始，每次循环迭代就加1，因此当循环结束时，countWeeks的值就是蟑螂体积超过房间体积所花费的星期总数了。

▲ 易犯错误：无限循环

程序中一种常见的错误是循环不会终止，一次次永远地执行它的循环体。不断地迭代它的主体而永不结束的循环称为无限循环（infinite loop）。通常，while循环或do-while循环中的某些语句会对一些变量的值进行修改，以使控制布尔表达式值为假。但是，如果没有用正确的方式修改变量，就可能会造成无限循环。要提供一个无限循环的例子，只要对你见过的一个循环进行少量修改就行了。

我们来考虑一种与图3-10中的程序略有不同的情况。假设你的家乡受到一种吃蟑螂的青蛙的侵扰。这些青蛙吃蟑螂的速度非常快，使蟑螂的实际数量在减少，因此，蟑螂的增长率为负数。要反映这种情况，就要将一个已定义常量的定义改成下列形式，并重新编译程序：

```
public static final double GROWTH_RATE = -0.05; //-5% per week
```

如果进行了这种修改，并运行程序，while循环就会成为一个无限循环（假设开始房间里只有数量相对较少的蟑螂）。这是因为蟑螂的总数及蟑螂的总体积。在不断地减少，因此下面所示的控制布尔表达式总会为真：

```
(totalBugVolume < houseVolume)
```

因此，循环永远也不会结束。

实际上有些无限循环不会永远运行下去，而是会在某些系统资源耗尽时，以一种不正常的状态终止程序。但对有些无限循环来说，如果不管它，它就会一直运行下去。要终止一个带有无限循环的程序，就要了解如何强制一个程序停止运行。对不同的操作系统来说，实现的方法是不同的。在很多系统（但不是所有的系统）中，都可以通过按Ctrl+C键来终止一个程序。

有时，程序员可能会有意地编写一个无限循环。例如，ATM机通常就是由一个带有无限循环的程序控制的，这个循环会不停地处理存款和取款事宜。但是，现在在用户自己编程时，无限循环就很可能是个错误。 △

自测题

18. 下列代码会产生什么样的输出？

```
int count = 0;
while (count < 5)
{
    System.out.println(count);
    count++;
}
System.out.println("count after loop = " + count);
```

19. while循环的主体可以执行0次吗？do-while循环的主体可以执行0次吗？
20. 下列代码会产生什么样的输出？

```
int count = 0;
do
{
    System.out.println(count);
    count++;
}while(count < 0);
System.out.println("count after loop = " + count);
```

21. 重新编写下列do-while循环，以获取不包含do-while循环的等效代码（keyboard是一个Scanner对象）。

```
int number;
do
{
    System.out.println("Enter a whole number:");
    number = keyboard.nextInt();
    System.out.println("You entered " + number);
}while(number > 0)
System.out.println("number after loop = " + number);
```

22. 下列代码会产生什么样的输出？

```
int count = 0;
while(count < 5)
{
    System.out.println(count);
    count--;
}
System.out.println("count after loop = " + count);
```

3.2.3 for语句

for语句是一种特殊的循环语句，可以使你很容易地将下面这样的伪代码转换成Java循环。

```
对从1~3的每个count值做下列动作:
    System.out.println(count);
System.out.println("Go");
```

在Java中，可以用下列（后面跟一条输出语句的）for语句来表示这种特殊的伪代码：

```
for (count = 1; count <= 3; count++)
    System.out.println(count);
System.out.println("Go");
```

上面的前两行代码是一条for语句，会产生下列输出：

```
1
2
3
```

for语句结束之后，最后一行代码会输出单词"Go"。

在for语句的这个例子中，循环体是语句

```
System.out.println(count);
```

循环体的迭代是由下面这行代码控制的：

```
for (count = 1; count <= 3; count++)
```

圆括号内3个表达式中的第一个count = 1，说明了在第一次执行循环体之前发生的情况；第三个表达式count++，是在循环体的每次迭代之后执行的；中间的表达式count <= 3，是一个布尔表达式，用来确定循环何时终止，它的工作方式与while循环中的控制布尔表达式的工作方式相同。因此，当count <= 3为真时，就执行循环体。换句话说，for语句

```
for (count = 1; count <= 3; count++)
    for循环体
```

等价于

```
count = 1;
while(count <= 3)
{
    for循环体
    count++;
}
```

for语句的语法如下所示：

for(*Initializing_Action*; *Boolean_Expression*; *Update_Action*)
　　Body_Statement

*Body_Statement*可以是一条单语句，如下例：

```
for (count = 1; count <= 3; count++)
    System.out.println(count);
```

但是，如图3-11所示，在更多的情况下，*Body_Statement*是复合语句，因此，更通用的for循环形式如下：

for (*Initializing_Action*; *Boolean_Expression*; *Update_Action*)
{
　　First_Statement
　　Second_Statement
　　…
　　Last_Statement
}

执行时，上述形式的for语句等效于下列代码：

Initializing_Action;
while (*Boolean_Expression*)
{
　　First_Statement
　　Second_Statement
　　…
　　Last_Statement
　　Update_Action
}

注意，for语句本质上就是某类while循环的另一种表示法。因此，和while循环一样，for语句的循环体也可以重复0次。

```
public class ForDemo
{
    public static void main(String[] args)
    {
        int countDown;

        for (countDown = 3; countDown >= 0; countDown--)
        {
            System.out.println(countDown);
            System.out.println("and counting.");
        }
        System.out.println("Blast off!");
    }
}
```

屏幕输出

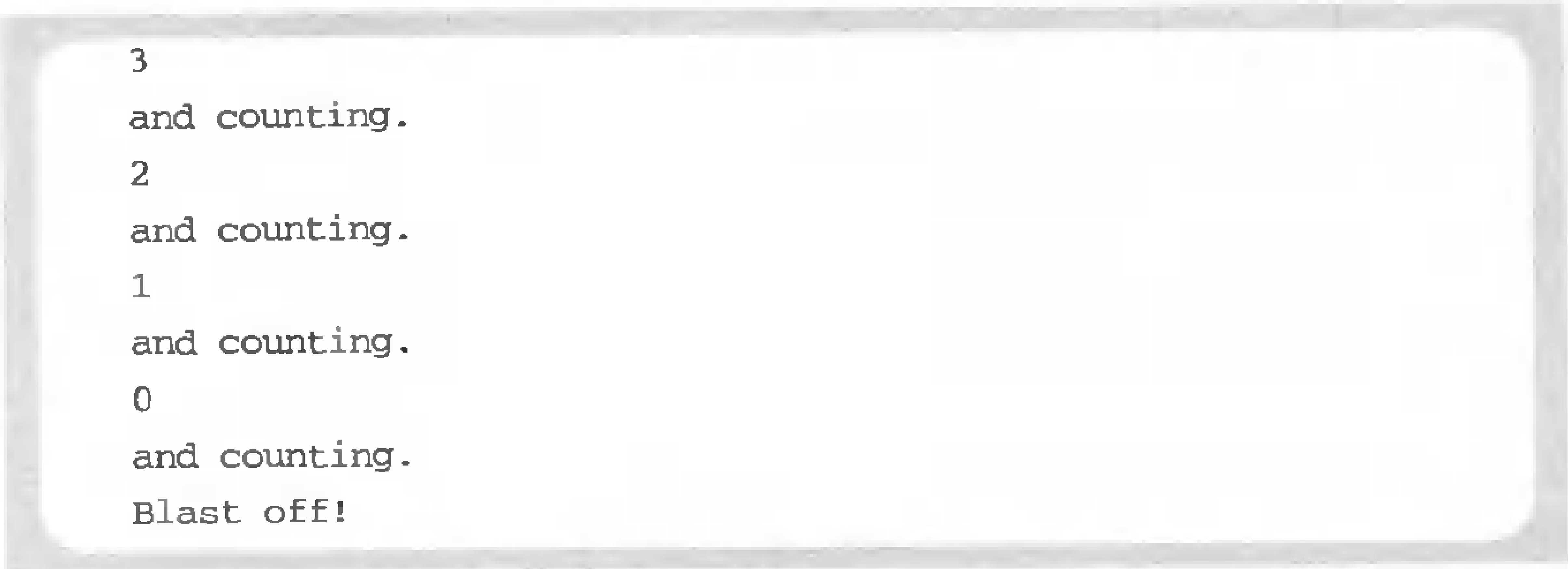


图3-11 for语句

图3-12描述的是for循环的语义（含义）。

for (*Initializing_Action*; *Boolean_Expression*; *Update_Action*)
    Body

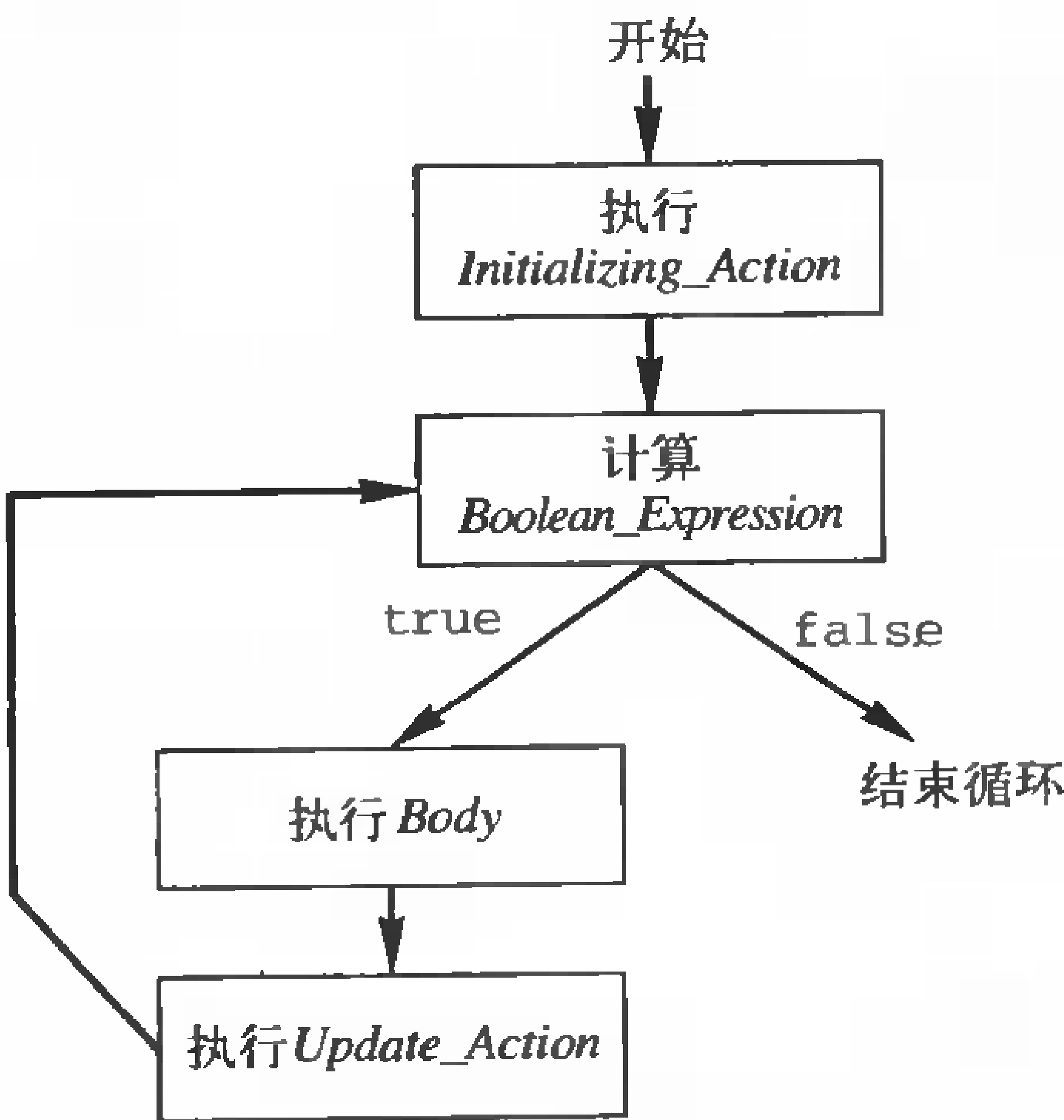


图3-12 for语句的语义

举例

```
for (countDown = 3; countDown >=0; countDown--
{
    System.out.println(countDown);
    System.out.println("and countin.");
}
```

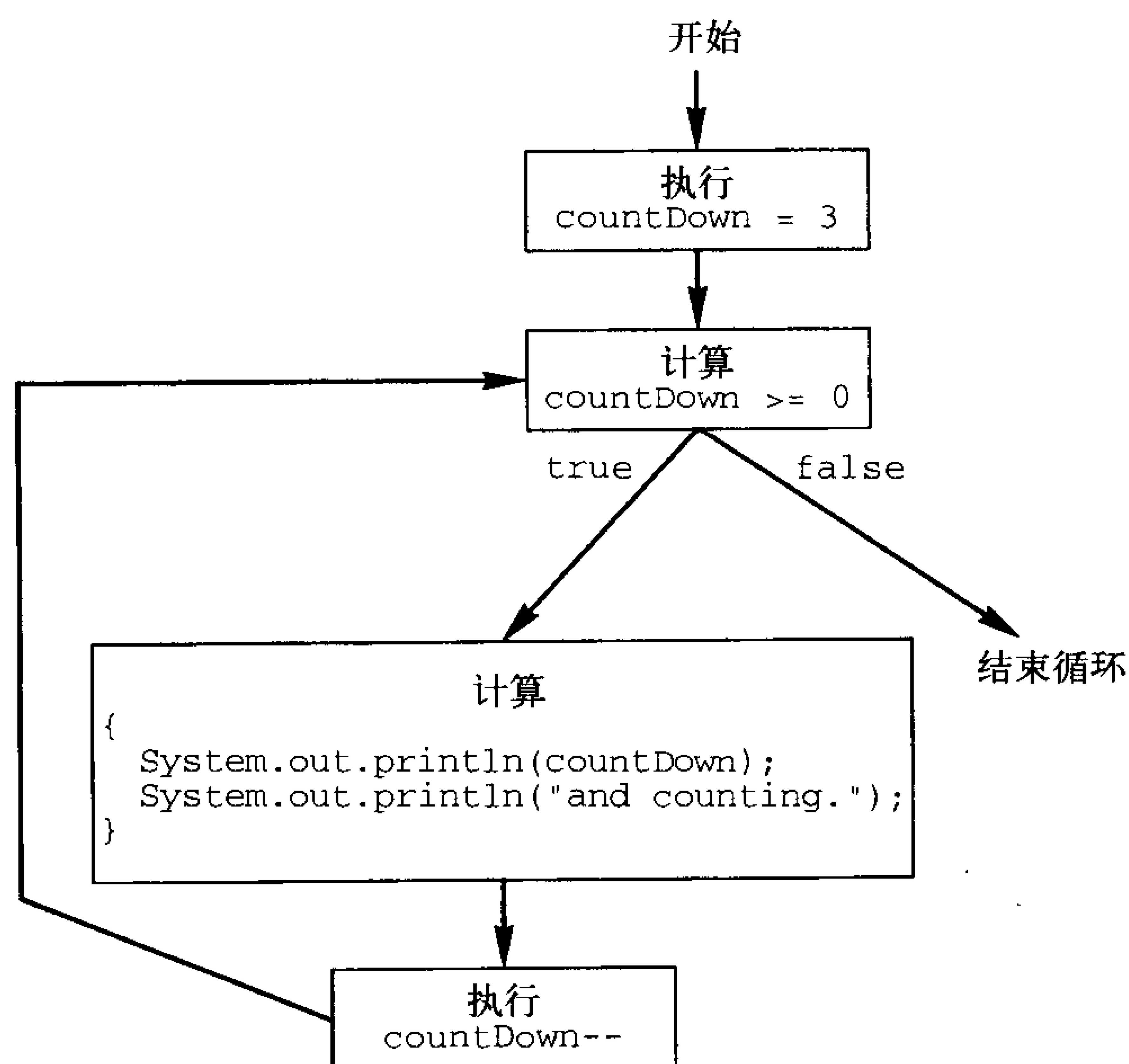


图3-12　（续）

快速参考：for语句

语法：

for (*Initializing_Action; Boolean_Expression; Update_Action*)
　　Body

*Body*可以是一条单语句，但更常见的是由一系列包含在花括号{}中的语句组成的复合语句。注意，圆括号中的3项是由两个，而不是3个分号隔开的。

本书中for循环的*Update_Action*通常只对一个变量的值进行修改。但在*Update_action*中是可以使用任意Java表达式的，因此，在这个表达式中使用的变量实际上可以多于或少于一个，而且，变量可以是任意类型的。*Initializing_Action*和*Boolean_Expression*中也可以包含多个变量。

举例：

```
for (next = 0; next <= 10; next = next + 2)
{
    sum = sum + next;
    System.out.println("sum now is " + sum);
}
```

3.2.4　for语句中的逗号（选读）

for循环可以执行多个初始化动作。要想执行一系列的初始化动作，只要像下面的例子这

样，用逗号将这些动作分隔开就行了：

```
for (n = 1, product = 1; n <= 10; n++)
    product = product * n;
```

这个for循环将n初始化为1，并将product也初始化为1。注意，我们是用逗号（而不是分号）来分隔这些初始化动作的。

不能用多个布尔表达式来测试一个for循环的结束。但可以用&&运算符将多个测试表达式拼接在一起，以形成一个更大的布尔表达式。

可以用逗号将多个更新动作拼接在一起，以拥有多个更新动作。这样有时会造成下面这种情况：for语句的主体为空，但仍然能做一些有用的事情。例如，可以将前面的for语句重写成下列语句：

```
for( n = 1, product = 1; n <= 10; product = product * n, n++)
```

实际上，我们已经将循环体变成更新动作的一部分了。但是，如果像这个for循环的前一版本那样，只对控制循环的变量使用更新动作，代码将具有更强的可读性。我们不提倡使用无主体的for循环，但很多程序员会认为它们"更聪明"。正如在下面易犯错误"循环语句中额外的分号"中指出的那样，程序员的错误也经常会造成无主体的for循环。

如果用其他具有通用逗号运算符的编程语言编过程序，要提醒你，在Java中，逗号运算符只能在for语句中使用。

▲ 易犯错误：循环语句中额外的分号

下列代码看起来很正常，而且编译和运行时也不会有错误信息。但是，它确实包含了一个错误。在继续往下读之前，看看你能否找到这个错误。

```
int product = 1, number;
for (number = 1; number <= 10; number++);
    product = product * number;
System.out.println(
          "Product of the numbers 1 through 10 is " + product);
```

如果将这段代码包含在一个程序中，并运行这个程序，输出如下：

```
Product of the numbers 1 through 10 is 11
```

现在你能看出什么地方出错了吗？在继续往下读之前，试着解释一下这个问题。

如果你在测试这个产生了让人困扰的输出的程序，它会把你搞糊涂的。很明显，for循环出了问题，但是，是什么问题呢？for循环应该将product的值设置为

```
1 * 2 *3 * 4 * 5 * 6 * 7 * 8 * 9 * 10
```

但它却将product的值设置成了11。怎么会这样呢？

出现了一个很小的打字错误。在for语句第一行的末尾处有一个额外的分号：

```
for (number = 1; number <= 10; number ++);
product = product * number;
```

这条for语句做了什么呢？末尾的分号意味着for语句的主体是空的。分号自身被当成了一条什么事也没做的语句（这条语句被称为空语句（empty statement或null statement）。这条带有额外分号的for语句等效于：

```
for (number = 1; number <= 10; number ++)
{
    //Do nothing.
}
```

因此，for语句的主体实际上被执行了10次；但每次执行时，除了将变量number的值加1之外，循环什么都没做。这样，当程序执行到语句product = product * number;时，number就等于11了。

记住，number开始时等于1，其值增加了10次，每次加1，这样就在初始的1上加了10，因此，值变成了11。

再来看看整段问题代码：

```
int product = 1, number;
for (number = 1; number <= 10; number++);
    product = product * number;
System.out.println(
        "Product of the numbers 1 through 10 is " + product);
```

执行了以for开头的那行代码之后，product的值为1，number的值为11。然后，就会执行下列赋值语句：

```
product = product * number;
```

这会将product的值置为1*11，因此，就像输出中显示的那样，product最终的值为11。要修正这个问题，只要将以for开头的那行代码末尾额外的分号删除就可以了。

在while循环中也会发生同样的问题。下面的while循环和刚才那个出问题的for循环有同样的问题，但结果更糟糕：

```
int product = 1, number = 1;
while(number <= 10);
{
    product = product * number;
    number++;
}
System.out.println("Product of the numbers 1 through 10 is " + product);
```

额外的分号结束了while循环，因此，while循环的主体是一条空语句。因为循环的主体是空语句，每次循环迭代时，什么都不会发生。因此，number的值也从来没改变过，那么条件number <= 10就总是为true（真）。因此循环是一个什么也不做，却会一直运行下去的无限循环！　△

自测题

23. 下列代码会产生什么样的输出？

```
int n;
for ( n = 1; n <= 4; n++)
    System.out.println(n);
```

24. 下列代码会产生什么样的输出？

```
int n;
for ( n = 1; n > 4; n++)
    System.out.println(n);
```

25. 下列代码会产生什么样的输出？

```
int n;
for ( n = 4; n > 0; n--)
    System.out.println(n);
```

26. 下列代码会产生什么样的输出？

```
int n;
for ( n = 4; n > 0; n--);
    System.out.println(n);
```

（这道题和前面那道题是不一样的。请仔细看看。）

27. 下列代码会产生什么样的输出？

```
double test;
for (test = 0; test < 3; test = test + 0.5)
    System.out.println(test);
```

28. 编写一条for语句，用来输出偶数2、4、6、8和10。输出应该将每个数字都输出到单独一行。先声明要用的所有变量。

■ Java提示：选择循环语句

假设确定了程序需要一个循环，该如何决定是选择while语句、do-while语句，还是for语句呢？我们可以给出一些通用的原则：除非能确定，对程序所有可能的输入来说，循环都至少要迭代一次，否则就不能使用do-while语句。如果知道循环总是至少会迭代一次，那么do-while循环可能会是个很好的选择。但是，循环主体需要迭代零次的可能性，比你想象的要多一些。在这些情况下，必须使用while语句或for语句。如果要做的是计算，且每次迭代时都要对某些数字值进行等量的修改，就可以考虑使用for语句。如果for语句不太好用，就用while语句。while语句总是最安全的选择。你可以很容易地用while语句来实现任意类型的循环，但有时某种其他选择会更好一些。

3.2.5 循环中的break语句

按照到目前为止的描述，while、do-while和for语句每次迭代时总会完成它们的整个循环体。但有时，你可能希望在循环体的中间终止一个循环。通过break语句就可以做到这一点。例如，图3-13中的程序读入一串购物数额，并计算出总额，看看用户花了多少钱。但是，用户有个100美元的限制，一旦总额达到（或超过）100美元，程序就立刻用一条break语句终止循环。执行break语句时，包含这个break的循环就终止，不再执行循环体的其余部分。break语句可以与while循环、do-while循环或for循环一起使用。

这条break语句和前面在switch语句中使用的break语句是一样的。如果这个循环包含在一个较大的循环内（或者这个循环在一条switch语句内部），break语句只会终止最内层的循环。类似地，如果break语句位于一个循环内的switch语句中，那么，break语句只会终止switch语句而不会终止循环。break语句只能终止最内层的循环或包含break语句的switch语句。

▲ 易犯错误：break语句的滥用

不包含break语句的循环结构简单、易于理解。在循环的顶端（或底端）有一个结束循环的测试，每次迭代都会执行到循环体的末尾。添加了break语句之后，对循环的理解会更困难一些。循环可能会由于循环开始（或底端）给出的条件而终止，也可能由于break语句而终止。有些循环迭代可能会执行到循环体的末尾，但有一次循环迭代可能会过早地结束。由于break语句所带来的复杂性，应该尽量避免在循环中使用break语句。有些权威主张永远都不要用break语句来终止一个循环，但几乎所有的编程专家都认为只能非常保守地使用break语句。△

快速参考：循环中的break语句

可以在switch语句或任意类型的循环语句中使用break语句。执行break语句时，包含这个break的循环（或switch语句）终止，不再执行循环体的其余部分。

```
import java.util.*;

public class BreakDemo
{
    public static void main(String[] args)
    {
        int itemNumber;
        double amount, total;
        Scanner keyboard = new Scanner(System.in);

        System.out.println("You may buy ten items, but");
        System.out.println("the total price must not exceed $100.");

        total = 0;
        for (itemNumber = 1; itemNumber <= 10; itemNumber++)
        {
            System.out.print("Enter cost of item #"
                                          + itemNumber + ": $");
            amount = keyboard.nextDouble();
            total = total + amount;
            if (total >= 100)
            {
                System.out.println("You spent all your money.");
                break;
            }
            System.out.println("Your total so far is $" + total);
            System.out.println("You may purchase up to "
                              + (10 - itemNumber) + " more items.");
        }
        System.out.println("You spent $" + total);
    }
}
```

屏幕对话示例

```
You may buy ten items, but
the total price must not exceed $100.
Enter cost of item #1:$90.93
Your total so far is $90.93
Your may purchase up to 9 more items.
Enter cost of item #2:$10.50
You spent all your money.
You spent $101.43
```

图3-13 用break语句终止一个循环

3.2.6 exit方法

有时程序可能会碰到一种情况，使继续执行程序变得毫无意义。在这种情况下，可以按如下所示的方法，用exit方法调用来终止程序：

```
System.exit(0);
```

一旦执行了前面的语句，就会立即终止Java程序。

例如：

```
if ( numberOfWinners == 0)
{
     System.out.println("Error: Dividing by zero.");
     System.exit(0);
}
else
{
     oneShare = payoff/numberOfWinners;
     System.out.println("Each winner will receive $" + oneShare);
}
```

这条语句通常会输出每个获胜者应得的份额。但如果获胜者的数目为零，就会出现被零除的现象，是非法操作。为了避免这种被零除的情况，程序查看了获胜者的数目是否为零，如果为零，就调用exit方法来终止程序。

作为实参传递给System.exit的数字0被返回给了操作系统。在很多情况下都可以使用任意数字，程序会以相同的方式运行。但大多数操作系统都用0来说明程序的正常终止，而用1来说明程序的非正常终止（和大多数人的猜想相反）。因此，如果System.exit语句正常地终止了程序，实参就应该为0。在这种情况下，*正常*（normal）意味着程序没有违反任何系统约定或其他重要约定，但并不意味着程序做了用户希望它做的事情。因此，通常用0作为实参。

快速参考：exit方法

调用exit方法会终止程序。exit方法调用的常用形式为

```
System.exit(0);
```

自测题

29. 下列代码会产生什么样的输出？

```
int n;
for ( n = 1; n <= 5; n++)
{
    if ( n == 3)
        break;
    System.out.println("Hello");
}
System.out.println("After the Loop.");
```

30. 下列代码会产生什么样的输出？

```
int n;
for ( n = 1; n <= 5; n++)
{
    if ( n == 3)
        System.exit(0);
    System.out.println("Hello");
}
System.out.println("After the Loop.");
```

31. 下列代码会产生什么样的输出？

```
int n;
for ( n = 1; n <= 3; n++)
```

```
{
    switch(n)
    {
        case 1:
            System.out.println("One.");
            break;
        case 2:
            System.out.println("Two.");
            break;
        case 3:
            System.out.println("Three.");
            break;
        default:
            System.out.println("Default case.");
            break;
    }
}
System.out.println("After the Loop.");
```

3.3 用循环编程

*以约失之者鲜矣！*①

——孔子，《论语·里仁》

循环通常可以分成3个部分：必须出现在循环之前的初始化语句，循环体，以及结束循环的机制。我们将在本节介绍每种循环组件的设计技巧。尽管初始化语句出现在循环体之前，但通常会很自然地先设计循环体，因此，首先介绍循环体。

3.3.1 循环体

设计循环体的一种方法是将你希望代码完成的动作序列写下来。比如，你可能希望循环执行下列动作：

```
向用户输出一些指令
初始化变量
将一个数字读入变量next
sum = sum + next;
输出当前数字及数字和
将另一个数字读入变量next
sum = sum + next;
输出当前的数字及数字和
将另一个数字读入变量next
sum = sum + next;
输出当前的数字及数字和
将另一个数字读入变量next
依次类推
```

然后在这个动作列表中查找重复模式。在这种情况下，重复模式为：

```
将另一个数字读入变量next
sum = sum + next;
```

① 意为，能约束自己，就很少会犯过失。—编者注

```
输出当前的数字及数字和
```

因此，用伪代码表示的循环体就包含前面这3个动作。完整的伪代码为：

```
向用户输出一些指令
初始化变量
将下列代码执行适当的次数：
{
   将一个数字读入变量next
   sum = sum + next;
   输出当前的数字及数字和
}
```

注意，重复模式不一定要从第一个动作开始。在执行循环之前或之后都可能有需要执行的动作。

快速参考：伪代码

在第1章介绍过，算法通常都是以伪代码书写的。**伪代码**（pseudocode）是自然语言和Java语言的混合体。使用伪代码时，只要用对你来说最简单的语言将算法的每个部分都写下来就行了。如果某个部分用自然语言表达更简单，就用自然语言；如果另一个部分用Java表达更简单，就用Java。下面这个简单的算法就是一个伪代码实例：

```
将另一个数字读入变量next
sum = sum + next;
将当前的数字及数字和输出
```

3.3.2 初始化语句

对于在3.3.1节设计的伪代码，每次执行下列循环体语句时，都会为变量sum赋一个值：

```
sum = sum + next;
```

特别要注意，循环第一次迭代时也是如此。因此，必须在循环开始之前将sum初始化为某个值。在试图为sum确定正确的初始值时，考虑一下你希望在一次循环迭代之后发生什么情况是很有帮助的。在当前设计的循环中，一次循环迭代之后，sum的值应该被设置成next的第一个值。sum + next经计算后等于next的唯一可能就是sum为0。这就意味着sum的值必须被初始化为0。因此，必须有一条变量初始化语句：

```
sum = 0;
```

循环中使用的其他变量只有一个next。对next执行的第一条语句为

```
将一个数字读入变量next。
```

这条语句为next赋了一个值，这样，就不需要在循环开始之前为next赋值了。因此，唯一需要进行初始化的变量就是sum，可以将伪代码重写如下：

```
向用户输出一些指令
sum = 0;
将下列代码执行适当的次数：
{
   将一个数字读入变量next
   sum = sum + next;
   将当前的数字及数字和输出
}
```

变量不总是被初始化为零的。为了说明这一点，来看另一个例子。假设循环要计算n个数的乘积，代码如下：

```
for (count = 1; count <= n; count++)
{
    将一个数字读入变量next
    product = product * next;
}
```

在这个例子中，假设所有的变量都是int类型的。

如果将变量product初始化为0，那么无论读入多少个数，乘了多少个数，product的值都仍然为0。因此，很显然0不是product的正确初始值。product的正确初始值为1。要说明1是正确的初始值，注意第一次执行循环时，希望将product设置为读入的第一个数。将product初始化为1就可以实现这一点。因此，带有正确初始化语句的循环为

```
product = 1;
for (count = 1; count <= n; count++)
{
    将一个数字读入变量next
    product = product * next;
}
```

3.3.3 结束循环

本节将介绍一些可以用来结束循环的标准技术。

幸运的话，程序会确切地知道循环开始之后，循环体要重复多少次。在这种简单的情况下，可以用for循环对循环体的迭代次数进行计数。例如，假设numberOfStudents中包含了一个班上学生的数量，而你想知道一门课程考试的平均分数。使用下面的代码可以很好地完成这项工作：

```
double next, average, sum = 0;
int count;
Scanner keyboard = new Scanner(System.in);
for (count = 1; count <= numberOfStudents; count++)
{
    next = keyboard.nextDouble();
    sum = sum + next;
}
if (numberOfStudents > 0)
    average = sum/numberOfStudents;
else
    System.out.println("No scores to average.");
```

注意，循环体中没有使用变量count。for循环机制只是用来从1到numberOfStudents计数，并将循环体重复那么多次。像这样，在循环开始之前就知道循环迭代次数的循环被称为**计数控制循环**（count-controlled loop）。计数控制循环不一定要作为for循环来实现，但这是实现它的最简单方式。还要注意，我们允许班上没有学生的可能性存在。在那种情况下，循环体迭代零次，if-else语句会防止被零除的情况出现。

结束一个循环最直接的方式就是询问用户是否该结束循环了。这种技术被称为**迭代前询问**（ask-before-iterating）。在循环体迭代总次数相当小的情况下，这种技术效果较好。例如，如果每个客户都只买几样东西，下面的代码效果较好：

```
do
{
    System.out.println("Enter price $");
    price = keyboard.nextDouble();
    System.out.print("Enter number purchased:");
    number = keyboard.nextInt();
    System.out.println(number + " items at $" + price);
    System.out.println("Total cost $" + price*number);
    System.out.println("Want to make another purchase?");
    System.out.println("Enter yes or no.");
    answer = keyboard.next();
}while(answer.equalsIgnoreCase("yes"));
```

在某些情况下，最好用while循环来实现。但是如果知道每个客户都希望至少进行一次循环迭代，使用do-while循环的效果也不错。

有时可以为那些很长的输入序列使用一个**标记值**（sentinel value）。标记值是用来标记输入结束的。它必须是一个与所有可能的实际输入值不同的值。例如，假设你需要一些代码来计算一次考试的最高分和最低分，并知道至少会有一个考试成绩存在。如果你知道没人会在考试中得负分，就可以请求用户以一个负数来标记分数序列结束。这个负数就是标记值。它不是一个考试成绩，只是一个结束标记。计算最高分和最低分的代码可能如下：

```
System.out.println("Enter scores for all students.");
System.out.println("Enter a negative number after");
System.out.println("you have entered all the scores.");
Scanner keyboard = new Scanner(System.in);
double max = keyboard.nextDouble();
double min = max; //The max and min so far are the first score.
double next = keyboard.nextDouble();
while (next >= 0)
{
    if (next > max)
        max = next;
    if (next < min)
        min = next;
    next = keyboard.nextDouble();
}
System.out.println("The highest score is " + max);
System.out.println("The lowest score is " + min);
```

一定要注意，不要用最后一个数字（标记值）来确定最低分（或最高分）。假设用户输入了下列分数：

```
100
90
10
-1
```

输出将为

```
The highest score is 100
The lowest score is 10
```

一定要注意，最低分为10，而不是-1。-1只是一个结束标志。

在3.4节中，将讨论另一种结束循环的方法，但是这里讨论的3种方法涵盖了你可能会遇到的大多数情况。

编程示例：嵌套循环

循环的主体中可以包含任意类型的语句。特别是，在一个较大循环语句的主体中可以包含一条循环语句。例如，图3-14中的程序用一个while循环算出了分数序列的平均值。程序请用户输入一个非负的分数序列，并用一个负的标记值来标识序列的结束。然后将这个while循环放在一个do-while循环内，这样用户就可以为另一次考试，以及再下一次考试重复整个过程，直到用户希望结束程序为止。

```
import java.util.*;

/**
 Determines the average of a list of (nonnegative) exam scores.
 Repeats for more exams until the user says she/he is finished.
*/
public class ExamAverager
{
    public static void main(String[] args)
    {
        System.out.println("This program computes the average of");
        System.out.println("a list of (nonnegative) exam scores.");
        double sum;
        int numberOfStudents;
        double next;
        String answer;
        Scanner keyboard = new Scanner(System.in);
        do
        {
            System.out.println();
            System.out.println("Enter all the scores to be averaged.");
            System.out.println("Enter a negative number after");
            System.out.println("you have entered all the scores.");
            sum = 0;
            numberOfStudents = 0;
            next = keyboard.nextDouble();
            while (next >= 0)
            {
                sum = sum + next;
                numberOfStudents++;
                next = keyboard.nextDouble();
            }
            if (numberOfStudents > 0)
                System.out.println("The average is "
                                     + (sum/numberOfStudents));
           else
                System.out.println("No scores to average.");
            System.out.println("Want to average another exam?");
            System.out.println("Enter yes or no.");
            answer = keyboard.next( );
        }while (answer.equalsIgnoreCase("yes"));
    }
}
```

图3-14 嵌套循环

屏幕对话示例

```
This program computes the average of
a list of (nonnegative) exam scores.

Enter all the scores to be averaged.
Enter a negative number after
you have entered all the scores.
100
90
100
90
-1
The average is 95.0
Want to average another exam?
Enter yes or no.
yes

Enter all the scores to be averaged.
Enter a negative number after
you have entered all the scores.
90
70
80
-1
The average is 80.0
Want to average another exam?
Enter yes or no.
no
```

图3-14 （续）

● 编程提示：避免在循环体内声明变量

注意，在图3-14中，我们将所有变量的声明都放在了程序的起始位置，这样，它们就位于外层do-while循环体之外了。如果我们将一些声明放在do-while循环内，那么，每执行一次do-while循环体，就会将这些声明重复一次。因为这样可能会在每次循环迭代时重新创建变量，所以根据编译器的编写方式，这样做的效率可能很低。有时候在循环体内声明一个变量是有意义的，但是如果可以很方便地将变量声明移到循环之外，通常会比较好。

自测题

32. 编写一个Java循环语句，将短语"One more time."向屏幕输出4次。还要给出所有必需的声明或初始化语句。

33. 给出一个Java循环语句，将变量result设置为2^5。用一个循环来实现这项功能，循环开始时result的值等于1，进行5次迭代，每次迭代将result的值乘2。还要给出所有必需的声明或初始化语句。

34. 下列代码会产生什么样的输出？

```
int count, innerCount;
for (count = 0; count <= 3; count++)
    for (innerCount = 0; innerCount < count; innerCount++)
        System.out.println(innerCount);
```

35. 给出一个Java循环语句，读入一个double类型的数字序列，然后输出它们的平均值。数字全都大于或等于1.0。数字序列以一个标记值结束，且必须指定该标记值。还要给出所有必需的声明或初始化语句。

3.3.4 循环错误

与在开始使用循环之前见过的那些较简单程序相比，带有循环的程序更容易出现错误。幸好，在设计循环时最容易犯的错误类型有一个模式，这样我们就可以告诉你应该注意些什么问题了。此外，还有一些可以用来定位并修复程序中错误的标准技术。

最常见的两种循环错误类型为非预期的无限循环和循环次数差一次（off-by-one）错误。下面按顺序讨论。

我们已经介绍过无限循环了。但是，关于无限循环，还需要强调一个细节。一个循环，对某些输入值来说也许能终止，但对其他一些值来说也许会是个无限循环。仅仅用某些程序输入值对循环进行了测试，并发现循环终止了，并不意味着对其他一些输入值来说，它就不会是无限循环了。

例如，你有一个朋友，他的支票账户透支了。银行每个月会对余额为负的账户收取一笔罚金。你的朋友需要一个程序告诉他，如果他每个月存入固定数额的款项，要多长时间才能使账户余额非负。你设计了下列代码：

```
count = 0;
while (balance < 0)
{
    balance = balance - penalty;
    balance = balance + deposit;
    count++;
}
System.out.println("You will have a nonnegative balance in " + count + " months.");
```

将这段代码放入一个完整的程序中，并用一些合理的值对代码进行测试，比如罚金为15美元，而存款额为50美元，程序运行得很好。于是你把程序给了你的朋友，朋友运行之后发现程序陷入了无限循环。出了什么问题呢？因为你的朋友决定每个月只存入10美元。但是，当账户为负时，银行每个月要收取15美元的罚金。这样，即使存款了，每个月的账户余额也只会变成一个越来越大的负数。

看起来好象不会发生这种事情。你的朋友不会犯这么愚蠢的错误。但实际是会发生这种事情的。人们有时候是很粗心的。修正这个错误的一种方法是，添加一些代码，用来测试循环是否是无限的。例如，可以将代码改成下列形式：

```
if (payment <= penalty)
    System.out.println("payment is too small.");
else
{
    count = 0;
    while (balance < 0)
    {
        balance = balance - penalty;
        balance = balance + payment;
        count++;
```

```
    }
    System.out.println("You will have a nonnegative balance in "
                                          + count + " months.");
}
```

另一种常见的循环错误类型是**循环次数差一次错误**。这个错误表示循环将循环体多重复或少重复运行了一次。设计控制布尔表达式时的粗心可能会导致这种类型的错误。例如，如果在应该使用小于或等于时使用了小于运算符，循环体的迭代次数就很容易出错。

对循环的控制布尔表达式来说，另一种常见问题与测试相等性的==的用法有关。这种相等性测试对整数和字符的测试都很令人满意，但对浮点数就不那么可靠了。这是因为浮点数是近似值，而==测试的是精确的相等性。这样一个测试的结果是无法预料的。对浮点数进行比较时，通常会使用包含小于或大于号在内的运算符，如<=；而不会使用==或!=。用==或!=对浮点数进行测试可能会产生循环次数差一次错误、非预期的无限循环，甚至一些其他类型的错误。

对循环次数差一次错误来说，一个很大的危险是这种错误很容易被忽略。如果循环多迭代或少迭代了一次，结果看起来可能是合理的，但结果的偏差却足以在以后引起麻烦。随时将循环结果与采用其他手段得到的正确结果进行比较，以便对循环次数差一次错误进行专门的检查，比如，对简单的情况，就可以用纸笔计算得到结果。

记住：随时进行重复测试

每当在程序中发现一个错误，并"修正"时，都要对程序进行重复测试。程序中可能还存在其他错误，有时，"修正"也可能会引发新的错误。

3.3.5 跟踪变量

如果程序的表现不太正常，但你又看不出什么地方出错了，可以采取的最好措施就是对某些关键变量进行跟踪。**跟踪变量**（tracing variable）意味着在程序运行时查看变量值的变化。通常程序不会在每次变量值发生变化时都将其输出，但看看变量是如何变化的对调试程序是有帮助的。

很多系统都有内建的实用工具，使你不需要对程序进行任何修改，就可以很容易地跟踪变量。不同系统上的调试系统多少会有些不同。如果有这样一个调试工具，值得了解一下如何使用。如果没有这样的调试工具，只要在程序中插入一些额外的临时输出语句就可以跟踪变量了。例如，假设想跟踪下列（包含一个错误的）代码中的变量：

```
count = 0;
while (balance < 0)
{
    balance = balance + penalty;
    balance = balance - deposit;
    count++;
}
System.out.println("Nonnegative balance in " + count + " months.");
```

可以通过添加下列输出语句来跟踪变量：

```
count = 0;
System.out.println("count == " + count); //trace
System.out.println("balance == " + balance); //trace
System.out.println("penalty == " + penalty); //trace
System.out.println("deposit == " + deposit); //trace
while (balance < 0)
{
    balance = balance + penalty;
    System.out.println("balance + penalty == " + balance);//trace
    balance = balance - deposit;
    System.out.println("balance - deposit == " + balance);//trace
    count++;
    System.out.println("count == " + count); //trace
}
System.out.println("Nonnegative balance in "
                        + count + " months.");
```

发现代码中的错误并将其修复之后，可以将这些跟踪语句删除。

在前面的例子中插入所有这些跟踪语句看起来可能很麻烦，但工作量并不大。如果愿意，你可以先试着跟踪部分变量，看是否能够给出足够的信息，以找到问题所在。但是，从一开始就跟踪所有或绝大部分的变量，通常是最快的。

自测题

36. 3.3.5节中的代码有什么错误？

37. 在下列代码中添加一些适当的输出语句，以便对所有变量进行跟踪：

```
int n, sum = 0;
for (n = 1; n < 10; n++)
    sum = sum + n;
System.out.println("1 + 2 + ··· + 9 + 10 == " + sum);
```

38. 下列代码有什么错误？这类循环错误被称作什么？

```
int n, sum = 0;
for (n = 1; n < 10; n++)
    sum = sum + n;
System.out.println("1 + 2 + ··· + 9 + 10 == " + sum);
```

3.4 boolean类型

真相就在那里。

——电视节目“X档案”的字幕

能够区分假象和真实的人必须对什么是真实、什么是假象有足够的了解。

——本尼狄克·斯宾诺莎，荷兰哲学家，《伦理学》

boolean类型是一种基本类型，就像类型int、double和char一样。就像其他这些类型一样，你也可以拥有boolean类型的表达式、boolean类型的值、boolean类型的常量和boolean类型的变量。但是，boolean类型的值只有两个：true和false。在程序中，可以像使用2和3.45这样的数字常量，以及'A'这样的字符常量一样，使用true和false这两个值。

布尔变量还可以用来使程序更易读。例如，程序中可能会包含下列语句，其中

systemsAreOK是一个boolean变量，如果发射系统准备好了，其值为true：

```
if (systemsAreOK)
    System.out.println("Initiate launch sequence.");
else
    System.out.println("Abort launching sequence.");
```

如果不使用boolean变量，上面的代码读起来很可能如下所示。

```
if ((temperature <= 100) && (thrust >= 12000)
                         && (cabinPressure > 30))
    System.out.println("Initiate launch sequence.");
else
    System.out.println("Abort launching sequence.");
```

很显然，使用boolean变量的第一种版本更容易理解。

当然，程序需要通过某种方式来设置boolean变量systemsAreOK的值，这很容易。

3.4.1 布尔表达式和布尔变量

对布尔表达式的计算会得到true或false这两个值之一。例如，下面代码中的表达式number > 0就是一个布尔表达式：

```
if (number > 0)
    System.out.println("The number is positive.");
else
    System.out.println("The number is negative or zero.");
```

如果计算number>0得到的值为true，输出的就是"The number is positive."；如果计算number > 0得到的值为false，输出的就是"The number is negative or zero."。在if-else语句这样的上下文中，number > 0 这样的布尔表达式的含义是比较好理解的。但在用boolean变量编程时，需要考虑的是或多或少没有上下文的布尔表达式。不需要引用任何if-else语句、while循环或其他在本节之前曾见过的上下文，就可以对布尔表达式进行计算，并产生一个为true或false的值。

可以用赋值语句将一个布尔表达式的值赋给一个boolean变量，赋值方法与用赋值语句设置int变量或任何其他类型变量的值的方法相同。例如，下列代码会将boolean变量isPositive的值设置为false：

```
int number = -5;
boolean isPositive;
isPositive = (number > 0);
```

如果愿意，可以按下列方式将后两行合并：

```
boolean isPositive = (number > 0);
```

圆括号不是必需的，但它们确实使程序更易读一些。

一旦boolean变量有了值，就可以像使用任何其他布尔表达式那样使用这个boolean变量了。例如：

```
boolean isPositive = (number > 0);
if (isPositive)
    System.out.println("The number is positive.");
else
    System.out.println("The number is negative or zero.");
```

等效于

```
if (number > 0)
    System.out.println("The number is positive.");
else
    System.out.println("The number is negative or zero.");
```

当然，这只是一个小例子。通常不会像第一个例子那样写程序，但如果像下列代码那样，number的值会发生变化，那么布尔表达式的值也会随着发生变化，这时，就应该使用与第一个例子类似的方法。下列代码可能是一个彩票计算程序的一部分：

```
System.out.println("Enter your number:");
Scanner keyboard = new Scanner(System.in);
number = keyboard.nextInt();
boolean isPositive = (number > 0)
while (number > 0)
{
    System.out.println("Wow!");
    number = number - 1000;
}
if (isPositive)
    System.out.println("Your number is positive.");
else
    System.out.println("Sorry, your number is not positive.");
System.out.println("Only positive numbers can win.");
```

可以用同样的方法来使用更复杂的布尔表达式。例如，如果systemsAreOK是一个boolean类型的变量，可以用下列方法为其赋值：

```
systemsAreOK = (temperature <= 100) && (thrust >= 12000)
                                    && (cabinPressure > 30);
```

● 编程提示：布尔变量的命名

命名布尔变量时，要选择一个在布尔表达式的值为true时也为真的表述方式，比如isPositive和systemsAreOK等。这样，将其用在while循环、if-else语句或其他控制语句中时，你就很容易理解这个布尔表达式的含义了。要避免使用那些不能很清楚地描述变量值含义的名字。不要使用numberSign、systemStatus这样的名字。

3.4.2 优先级规则

Java计算布尔表达式时所使用的策略，与计算算术表达式时所使用的策略相同。例如：

```
(score >= 80) && (score < 90)
```

假设score的值为95。计算第一个子表达式(score >= 80)得true，计算第二个子表达式(score < 90)得false。因此整个表达式可以缩写成如下形式：

```
true && false
```

计算机会根据一些规则将true和false的值结合起来，图3-15给出了这些称为**真值表**（truth table）的规则。据此，对前面表达式进行计算得到的值为false。

书写布尔表达式或算术表达式时，最好用圆括号来说明运算的顺序。但是，如果省略了圆括号，计算机会按照图3-16中所示的**优先级规则**（precedence rule）确定的顺序来执行这些运算。(图3-16显示了一段时间内可能用到的所有运算符。附录B给出了更完整的优先级规则列表。) 列表中上方列出的运算符具有**较高优先级**（higher precedence）。在没有圆括号指明运

算顺序的情况下，计算机在决定先执行两个运算符中的哪一个时，会在执行优先级较低的运算之前执行优先级较高的运算。有些运算符具有相同的优先级，运算顺序就由运算符从左至右的顺序来决定。相同优先级的二元运算符按照从左至右的顺序执行。相同优先级的一元运算符是按照从右至左的顺序执行。

&&（与）

*A*的值	*B*的值	*A*&&*B*的结果
true	true	true
true	false	false
false	true	false
false	false	false

||（或）

*A*的值	*B*的值	*A*||*B*的结果
true	true	true
true	false	true
false	true	true
false	false	false

!（非）

*A*的值	!（*A*）的结果
true	false
false	true

图3-15　布尔运算符的真值表

最高优先级

第一：一元运算符+、-、++、--和!
第二：二元算术运算符*、/、%
第三：二元算术运算符+、-
第四：布尔运算符<、>、<=、>=
第五：布尔运算符==、!=
第六：布尔运算符&
第七：布尔运算符|
第八：布尔运算符&&
第九：布尔运算符||

最低优先级

图3-16　优先级规则

下面看一个例子（回想一下，一元运算符只有一个实参——就是使用它的那个实参。二元运算符有两个实参。）下列代码的形式相当不好，但对计算机来说没有任何问题，用优先级规则来计算这段代码是很好的练习：

```
score < min/2 - 10 || score > 90
```

对表达式中所有的运算符来说，除法运算符具有最高优先级，因此，首先执行除法运算：

```
score < (min/2) - 10 || score > 90
```

对表达式中其余的运算符来说，减法运算符具有最高优先级，因此接下来要执行这个运算：

```
score < ((min/2) - 10) || score > 90
```

对表达式中其余的运算符来说，>和<运算符具有最高优先级，因此接下来要执行这些运算。因为>和<运算符具有相同的优先级，要按照从左至右的顺序来执行：

```
(score < ((min/2) - 10)) || (score > 90)
```

通过使用优先级规则生成了一个完全括号版本的表达式。对计算机来说，这两个表达式是等价的。

为了使算术及布尔表达式更易懂，编程时应该使用一些圆括号。但简单的&&或||串（而不是两者的混合）是个可以安全地省略圆括号的地方。比如，即使省略了一些圆括号，下列代码仍然具有很好的形式：

```
(temperature > 95) || (rainFall > 20) || (humidity >= 60)
```

Java处理||和&&的方式只比你到目前为止所见过的处理方式稍复杂一点儿。考虑下列布尔表达式：

```
(score > 90) || (assignmentsDone > 8)
```

假设score的值为95。在这种情况下，不论assignmentsDone的值是多少，布尔表达式的值都为true。这是因为true||true和true||false都得true。因此，不论assignmentsDone > 8是true还是false，整个表达式的值一定都为true。Java就是用这种方法来计算以||或&&连接起来的表达式的。它先计算第一个子表达式，如果其中包含了足够确定整个表达式值的信息，就不再计算第二个子表达式了。因此，在这个例子中，Java是绝不会去计算表达式assignmentsDone > 8的值。这种只按其所需计算部分表达式的方法被称为**短路求值**（short-circuit evaluation），是Java对&&和||使用的计算方式。短路求值有时也被称为**惰性求值**（lazy evaluation）。

现在，来看一个使用了&&的例子，将布尔表达式放在一条if-else语句中，给它一些上下文：

```
if ((assignmentsDone > 0)
                    && ((totalScore/assignmentsDone) > 60))
    System.out.println("Good work.");
else
    System.out.println("Work harder.");
```

假设assignmentsDone的值为0。那么第一个子表达式为false。因为false && true和false && false都为false，所以无论第二个子表达式为true还是false，整个表达式都为false。因此，Java就不再对第二个子表达式(totalScore/assignmentsDone)>60进行计算了。在这种情况下，是否计算第二个子表达式会有很大的区别，因为第二个子表达式中包含了被零除的情况。如果Java尝试计算第二个子表达式，就会生成一个运行时错误。通过使用短路求值，Java预防了一次运行时错误。

Java也允许你请求**完全求值**（complete evaluation）。在完全求值中，当两个表达式由“与”或“或”连接起来时，*总是会计算两个子表达式的值*，然后通过真值表得到最终表达式的值。要在Java中进行完全求值，就要用&而不是&&进行“与”，用|取代||进行“或”运算。

在大多数情况下，短路求值和完全求值都会给出相同的结果，但正如你刚才看到的，有时候短路求值能够避免运行时错误。也有一些情况更适合使用完全求值，但在本书中不会使用这些技术，因此，通常会用&&和||来进行短路求值。

3.4.3 布尔值的输入和输出

可以将boolean类型的值true和false作为输入和输出，用法与int和double等基本类型的值相同。例如，考虑下列来自一个Java程序的代码段：

```
boolean booleanVar = false;
System.out.println(booleanVar);
System.out.println("Enter a boolean value:");
Scanner keyboard = new Scanner(System.in);
booleanVar = keyboard.nextBoolean();
System.out.println("You entered " + booleanVar);
```

这段代码会生成下列对话：

```
false
Enter a boolean value:
true
You entered true
```

正如你从这个例子中看到的，Scanner类有一个名为nextBoolean的方法，可以读入单个boolean值。对这种输入方法来说，可以用大写或小写字母，或者两者的组合来拼写true和false。在Java程序中，常量true或false必须全部用小写字母来拼写；但输入方法nextBoolean要更宽容一些。

案例分析：用布尔变量来结束循环

在这个案例分析中，不是要解决一个完整的问题，而是要为一个经常发生的子任务设计一个循环，并将其放入一个演示程序中。这样，你会逐渐熟悉布尔变量最常见的一种用法。

你设计的循环要读入一个数字列表，并计算出列表中所有数字之和。你知道所有的数字都是非负的。例如，这些数字可能是一个编程团队中每个人工作的小时数列表。因为没人会工作负的小时数，所以所有的数字都是非负的，因此，可以用一个负数作为标记值来标识列表的结束。对这项任务来说，所有数字都是整数，但其他类型的数字，甚至读入的非数字数据，都可以使用相同的技术。

先用伪代码设计循环，可以使你对存在的问题以及可能的解决方法有更好的理解，因此，设计了下列伪代码：

```
int sum = 0;
为列表中每个数字做下列事情
if (数字为负)
使这次循环迭代成为最后一次
else
    sum = sum + the number;
```

由于序列的结束是用一个负数来标识的，所以可以将伪代码改进成下列形式：

```
int next, sum = 0;
while (还有需要读入的数字)
{
    next = keyboard.nextInt();
    if (next < 0)
    使这次循环迭代成为最后一次
    else
        sum = sum + next;
}
```

将这段伪代码转换成Java代码的方式有很多。刚刚学习了boolean类型变量，因此，你决定尝

试着使用它们（你会发现这是个正确的决定）。Boolean类型变量很好的一点是它读起来可以像个英文句子一样。因此，你决定尝试使用一个名为thereAreNumbersLeftToRead的boolean类型变量。只要声明这个boolean类型变量，并用其替代短语“还有需要读入的数字”就行了。这样就生成了下列代码：

```
int next, sum = 0;
boolean thereAreNumbersLeftToRead;
初始化变量thereAreNumbersLeftToRead
while (thereAreNumbersLeftToRead)
{
     next = keyboard.nextInt();
     if (next < 0)
     使这次循环迭代成为最后一次
     else
          sum = sum + next;
}
```

现在，完成这个循环以生成可以工作的Java代码，就很简单了。可以用一种显而易见的方式对短语“使这次循环迭代成为最后一次”翻译。当boolean变量thereAreNumbersLeftToRead的值为false时，循环就会终止。因此，终止循环的方式就是将thereAreNumbersLeftToRead设置为false。因此，可以将“使这次循环迭代成为最后一次”翻译成

```
thereAreNumbersLeftToRead = false;
```

剩下来要做的事就是要确定布尔变量thereAreNumbersLeftToRead的初始值。即使数字列表为空，至少也要读入一个标记值，因此循环体必须至少执行一次。所以，为了启动循环，thereAreNumbersLeftToRead必须为true，即它必须被初始化为true。这样，就会生成下列代码：

```
int next, sum = 0;
boolean thereAreNumbersLeftToRead = true;
while (thereAreNumbersLeftToRead)
{
     next = keyboard.nextInt();
     if (next < 0)
          thereAreNumbersLeftToRead = false;
     else
          sum = sum + next;
}
```

循环结束时，变量sum中包含的是输入列表中的数字之和（不包含标记值）。

剩下的就是要将循环放到一个程序中去。如果觉得变量名thereAreNumbersLeftToRead有点太长，可以把它缩短成numbersLeft，于是产生成图3-17中所示的程序。

自测题

39. 下列语句会产生什么样的输出？

```
int number = 7;
boolean isPositive = (number > 0);
if (number > 0)
    number = -100;
if (isPositive)
    System.out.println("Positive.");
else
System.out.println("Not positive.");
```

40. 下列语句会产生什么样的输出？

```
System.out.println(false);
System.out.println(7 < 0);
System.out.println(7 > 0);
int n = 7;
System.out.println(n > 0);
```

41. 下列语句会产生什么样的输出?

```
System.out.println(true && false);
System.out.println(true || false);
System.out.println(true || (x > 0));
```

```
import java.util.*;

/**
 Illustrates the use of a boolean variable to control loop ending.
*/
public class BooleanDemo
{
    public static void main(String[] args)
    {
        System.out.println("Enter nonnegative numbers.");
        System.out.println("Place a negative number at the end");
        System.out.println("to serve as an end marker.");

        int next, sum = 0;
        boolean numbersLeft = true;
        Scanner keyboard = new Scanner(System.in);
        while (numbersLeft)
        {
            next = keyboard.nextInt();
            if (next < 0)
                 numbersLeft = false;
            else
                 sum = sum + next;
        }
        System.out.println("The sum of the numbers is" + sum);
    }
}
```

屏幕对话示例

```
Enter nonnegative numbers.
Place a negative number at the end
to serve as an end marker.
1 2 3 -1
The sum of the numbers is 6
```

图3-17 用布尔变量来结束一个循环

3.5 图形编程补充（选读）

一幅彩色图画抵得上一千幅黑白图画。

——对一条中国谚语的改编

本节将继续对第1章开始的绘画进行讨论。特别是，会介绍如何向applet图画中添加颜色，还会给出一个用循环和分支语句进行绘画的示例。

我们会用对另一种JOptionPane窗口类型的描述来结束本节的内容，这种JOptionPane窗口会提出一些答案为“是”或“否”的问题。要有效地使用这种类型的窗口，就要用到本章介绍的分支结构，因此在第2章没有介绍这种JOptionPane窗口。

如果你已经选择了跳过与JOptionPane有关的内容，也可以跳过本节的JOptionPane部分。以后的章节都没有用到本章或第2章中的JOptionPane资料。另一方面，如果你想学习有关JOptionPane的内容而略过有关applet的内容，也是可以的。

3.5.1 指定绘画颜色

当你在applet的paint方法定义内，用drawOval这样的方法画形状时，可以认为你的画是用一支可以改变颜色的笔画出来的。方法setColor可以改变画笔的颜色。

例如，图3-18所示的笑脸是一张黄色的脸，脸上有蓝色的眼睛和红色的嘴唇。除了颜色之外，这张笑脸和图1-6及图2-16中的笑脸基本相同，只是现在向笑脸上加了一个鼻子。图3-18中所示的方法setColor的用法很常规，但有一点是需要强调的。下列代码来自图3-18，画了一个以黄色填充的圆圈作为脸：

```
canvas.setColor(Color.YELLOW);
canvas.fillOval(X_FACE, Y_FACE, FACE_DIAMETER, FACE_DIAMETER);
```

```
import javax.swing.*;
import java.awt.*;

public class YellowFace extends JApplet
{
    public static final int FACE_DIAMETER = 200;
    public static final int X_FACE = 100;
    public static final int Y_FACE = 50;

    public static final int EYE_WIDTH = 10;
    public static final int EYE_HEIGHT = 20;
    public static final int X_RIGHT_EYE = 155;
    public static final int Y_RIGHT_EYE = 95;
    public static final int X_LEFT_EYE = 230;
    public static final int Y_LEFT_EYE = Y_RIGHT_EYE;

    public static final int NOSE_DIAMETER = 10;
    public static final int X_NOSE = 195;//Center of nose will be at 200
    public static final int Y_NOSE = 135;
```

图3-18 添加颜色

```
    public static final int MOUTH_WIDTH = 100;
    public static final int MOUTH_HEIGHT = 50;
    public static final int X_MOUTH = 150;
    public static final int Y_MOUTH = 175;
    public static final int MOUTH_START_ANGLE = 180;
    public static final int MOUTH_DEGREES_SHOWN = 180;

    public void paint(Graphics canvas)
    {
        //Draw face circle:
        canvas.setColor(Color.YELLOW);
        canvas.fillOval(X_FACE, Y_FACE, FACE_DIAMETER, FACE_DIAMETER);
        canvas.setColor(Color.BLACK);
        canvas.drawOval(X_FACE, Y_FACE, FACE_DIAMETER, FACE_DIAMETER);
        //Draw eyes:
        canvas.setColor(Color.BLUE);
        canvas.fillOval(X_RIGHT_EYE, Y_RIGHT_EYE, EYE_WIDTH, EYE_HEIGHT);
        canvas.fillOval(X_LEFT_EYE, Y_LEFT_EYE, EYE_WIDTH, EYE_HEIGHT);

          //Draw nose:
          canvas.setColor(Color.BLACK);
          canvas.fillOval(X_NOSE, Y_NOSE, NOSE_DIAMETER, NOSE_DIAMETER);
          //Draw mouth:
          canvas.setColor(Color.RED);
          canvas.drawArc(X_MOUTH, Y_MOUTH, MOUTH_WIDTH, MOUTH_HEIGHT,
                          MOUTH_START_ANGLE, MOUTH_DEGREES_SHOWN);
    }
}
```

首先画填充了黄色的实心圆，这样，其他画都会画在黄色圆之上。

得到的applet

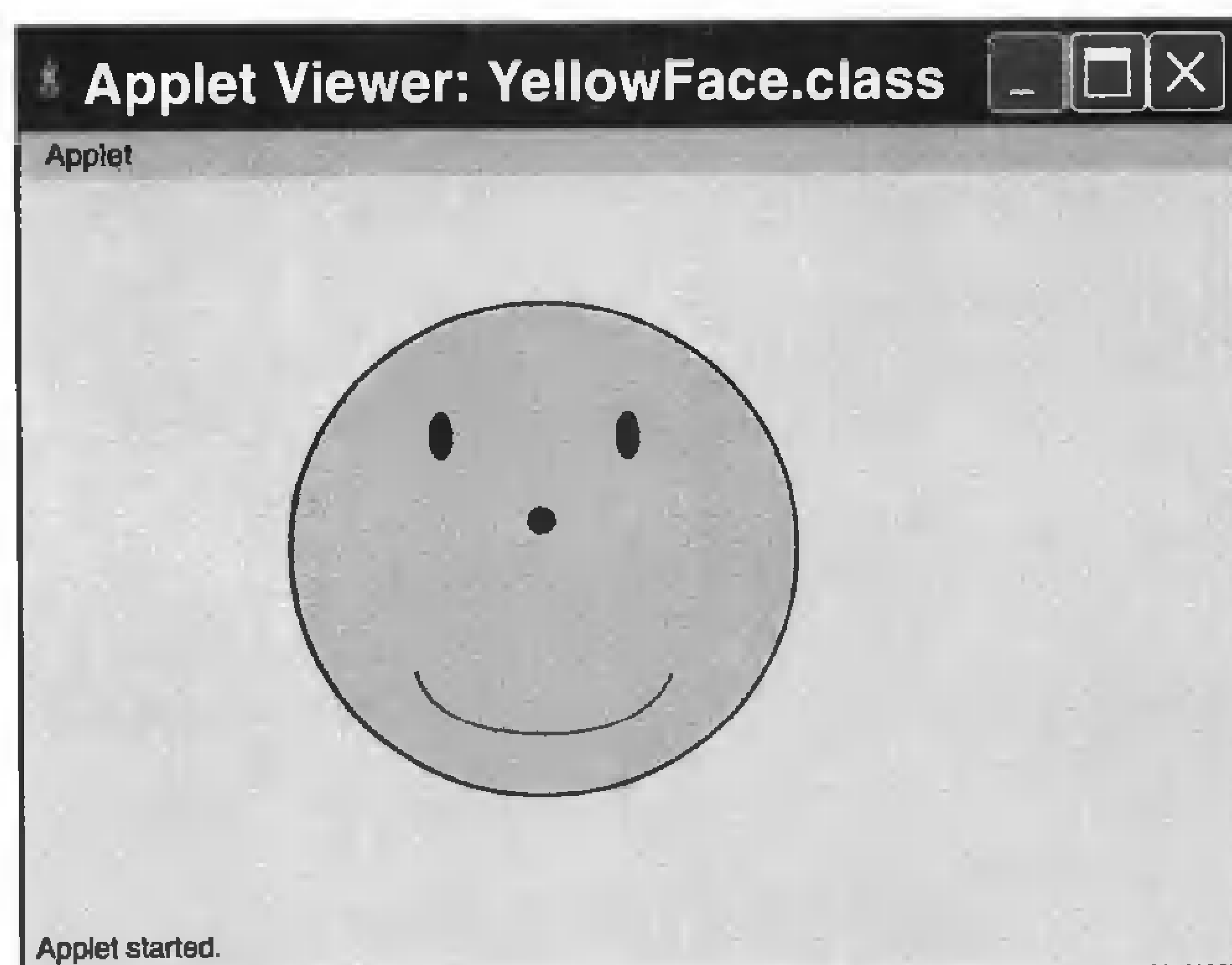


图3-18 （续）

注意，先画的是黄色的实心圆，这样，其他的画（比如眼睛）就会在黄色圆之上了。如果把画看成用画笔画的，那它就是按照paint方法中代码出现的顺序，将一种颜色画在另一种颜色之上。相反，如果我们最后画黄色的圆，那么，黄色圆就会在眼睛、鼻子和嘴巴之上，这样就只能看到黄色的圆了。

图3-19显示了已经定义好的颜色。也可以定义其他颜色，但在这里我们不对其进行深入讨论。

```
Color.BLACK          Color.MAGENTA
Color.BLUE           Color.ORANGE
Color.CYAN           Color.PINK
Color.DARK_GRAY      Color.RED
Color.GRAY           Color.WHITE
Color.GREEN          Color.YELLOW
Color.LIGHT_GRAY
```

图3-19　预定义的颜色

快速参考：setColor方法

用Graphics类的对象绘画时，可以通过调用setColor方法来设置绘画的颜色。之后，可以用再次调用setColor方法来修改这个指定的颜色，这样，一幅画面中就可以有多种颜色了。

举例：

```
canvas.setColor(Color.RED);
```

自测题

42. 假设你将图3-18中绘画命令的顺序改成下列形式。applet中的图画会发生变化吗？如果发生变化，会发生什么样的变化？

```
//Draw mouth:
canvas, setColor (Color. RED) ;
canvas.drawArc(X_MOUTH, Y_MOUTH, MOUTH_WIDTH, MOUTH_HEIGHT,
                MOUTH_START_ANGLE, MOUTH_DEGREES_SHOWN);
//Draw face circle:
canvas, setColor (Color. YELLOW) ;
canvas, fillOval(X_FACE, Y_FACE,
                      FACE_DIAMETER, FACE_DIAMETER) ;
canvas, setColor (Color. BLACK) ;
canvas, drawOval (X_FACE, Y_FACE,
                      FACE_DIAMETER,FACE_DIAMETER) ;
//Draw eyes:
canvas, setColor (Color. BLUE) ;
canvas, fillOval (X_RIGHT_EYE, Y_RIGHT_EYE,
                             EYE_WIDTH,EYE_HEIGHT);
canvas, fillOval (X_LEFT_EYE,
                      Y_LEFT_EYE,EYE_WIDTH,EYE_HEIGHT);
    //Draw nose:
    canvas, setColor (Color. BLACK) ;
    canvas, fillOval (X_NOSE, Y_NOSE,
                      NOSE DIAMETER, NOSE_DIAMETER) ;
```

43. 假设你将图3-18中绘画命令的顺序改成下列形式。applet中的图画会发生变化吗？如果发生变化，会发生什么样的变化？

```
//Draw face circle:
canvas, setColor (Color. YELLOW) ;
canvas, fillOval (X_FACE, Y_FACE,
                  FACE_DIAMETER, FACE_DIAMETER);
//Draw mouth:
canvas, setColor (Color. RED) ;
canvas.drawArc(X_MOUTH, Y_MOUTH, MOUTH_WIDTH, MOUTH_HEIGHT,
                 MOUTH_START_ANGLE, MOUTH_DEGREES_SHOWN) ;
canvas, setColor (Color. BLACK) ;
canvas, drawOval (X_FACE, Y_FACE,
                       FACE_DIAMETER, FACE_DIAMETER) ;
//Draw nose:
canvas, setColor (Color. BLACK) ;
canvas, fillOval (X_NOSE, Y_NOSE,
                       NOSE_DIAMETER,   NOSE_DIAMETER) ;
//Draw eyes:
canvas, setColor (Color. BLUE) ;
canvas, fillOval (X_RIGHT_EYE, Y_RIGHT_EYE,
                       EYE_WIDTH, EYE_HEIGHT);
canvas, fillOval (X_LEFT_EYE,
                       Y_LEFT EYE, EYE_WIDTH, EYE HEIGHT) ;
```

编程示例：画有多张脸孔的applet

图3-20中的applet包含了一系列脸孔。前5张脸孔在画成黄色和白色的脸孔间切换。这几张脸中，每张脸上的笑容都比前一张脸上的笑容更深一些。第6张脸在向用户飞吻，而第7张脸脸红了（也许是因为它对飞吻感到害羞了）。把这7张脸当成一个人的照片，每张都比前一张稍晚一些出现。（黄色和白色的交替只是想给出一些变化，没什么特别的含义。）

```
import javax.swing.*;
import java.awt.*;

public class MultipleFaces extends JApplet
{
    public static final int FACE_DIAMETER = 50;
    public static final int X_FACE0 = 10;
    public static final int Y_FACE0 = 5;

    public static final int EYE_WIDTH = 5;
    public static final int EYE_HEIGHT = 10;
    public static final int X_RIGHT_EYE0 = 20;
    public static final int Y_RIGHT_EYE0 = 15;
    public static final int X_LEFT_EYE0 = 45;
    public static final int Y_LEFT_EYE0 = Y_RIGHT_EYE0;

    public static final int NOSE_DIAMETER = 5;
    public static final int X_NOSE0 = 32;
    public static final int Y_NOSE0 = 25;
```

图3-20 使用循环和分支的applet

```
    public static final int MOUTH_WIDTH = 30;
    public static final int MOUTH_HEIGHT0 = 0;
    public static final int X_MOUTH0 = 20;
    public static final int Y_MOUTH0 = 35;
    public static final int MOUTH_START_ANGLE = 180;
    public static final int MOUTH_DEGREES_SHOWN = 180;

    public void paint(Graphics canvas)
    {
        int i;
        for (i = 0; i < 5; i++)
        {//Draw one face:
            //Draw face circle:
            if (i%2 == 0)//if i is even
            {//Make face yellow
                canvas.setColor(Color.YELLOW);
                canvas.fillOval(X_FACE0 + 50*i, Y_FACE0 + 30*i,
                            FACE_DIAMETER, FACE_DIAMETER);
            }
            canvas.setColor(Color.BLACK);
            canvas.drawOval(X_FACE0 + 50*i, Y_FACE0 + 30*i,
                            FACE_DIAMETER, FACE_DIAMETER);
            //Draw eyes:
            canvas.setColor(Color.BLUE);
            canvas.fillOval(X_RIGHT_EYE0 + 50*i, Y_RIGHT_EYE0 + 30*i,
                                EYE_WIDTH, EYE_HEIGHT);
            canvas.fillOval(X_LEFT_EYE0 + 50*i, Y_LEFT_EYE0 + 30*i,
                                EYE_WIDTH, EYE_HEIGHT);
            //Draw nose:
            canvas.setColor(Color.BLACK);
            canvas.fillOval(X_NOSE0 + 50*i, Y_NOSE0 + 30*i,
                                NOSE_DIAMETER, NOSE_DIAMETER);
            //Draw mouth:
            canvas.setColor(Color.RED);
            canvas.drawArc(X_MOUTH0 + 50*i, Y_MOUTH0 + 30*i,
                    MOUTH_WIDTH, MOUTH_HEIGHT0 + 3*i,
                    MOUTH_START_ANGLE, MOUTH_DEGREES_SHOWN);
        }
        //i == 5

        //Draw kissing face:
        //Draw face circle:
        canvas.setColor(Color.BLACK);
        canvas.drawOval(X_FACE0 + 50*i, Y_FACE0 + 30*i,
                            FACE_DIAMETER, FACE_DIAMETER);
        //Draw eyes:
            canvas.setColor(Color.BLUE);
```

循环体最后一次迭代之后，会对i的值进行最后一次运算。

图3-20 （续）

```
            canvas.fillOval(X_RIGHT_EYE0 + 50*i, Y_RIGHT_EYE0 + 30*i,
                            EYE_WIDTH, EYE_HEIGHT);
            canvas.fillOval(X_LEFT_EYE0 + 50*i, Y_LEFT_EYE0 + 30*i,
                            EYE_WIDTH, EYE_HEIGHT);
            //Draw nose:
            canvas.setColor(Color.BLACK);
            canvas.fillOval(X_NOSE0 + 50*i, Y_NOSE0 + 30*i,
                            NOSE_DIAMETER, NOSE_DIAMETER);
            //Draw mouth in shape of a kiss:
            canvas.setColor(Color.RED);
            canvas.fillOval(X_MOUTH0 + 50*i + 10, Y_MOUTH0 + 30*i,
                            MOUTH_WIDTH - 20, MOUTH_WIDTH - 20);
            //Add text:
            canvas.drawString("Kiss, Kiss.",
                      X_FACE0 + 50*i + FACE_DIAMETER, Y_FACE0 + 30*i);

            //Draw blushing face:
            i++;
            //Draw face circle:
            canvas.setColor(Color.PINK);
            canvas.fillOval(X_FACE0 + 50*i, Y_FACE0 + 30*i,
                            FACE_DIAMETER, FACE_DIAMETER);
            canvas.setColor(Color.BLACK);
            canvas.drawOval(X_FACE0 + 50*i, Y_FACE0 + 30*i,
                            FACE_DIAMETER, FACE_DIAMETER);
            //Draw eyes:
            canvas.setColor(Color.BLUE);
            canvas.fillOval(X_RIGHT_EYE0 + 50*i, Y_RIGHT_EYE0 + 30*i,
                            EYE_WIDTH, EYE_HEIGHT);
            canvas.fillOval(X_LEFT_EYE0 + 50*i, Y_LEFT_EYE0 + 30*i,
                            EYE_WIDTH, EYE_HEIGHT);
            //Draw nose:
            canvas.setColor(Color.BLACK);
            canvas.fillOval(X_NOSE0 + 50*i, Y_NOSE0 + 30*i,
                            NOSE_DIAMETER, NOSE_DIAMETER);
            //Draw mouth:
            canvas.setColor(Color.RED);
            canvas.drawArc(X_MOUTH0 + 50*i, Y_MOUTH0 + 30*i, MOUTH_WIDTH,
                            MOUTH_HEIGHT0 + 3*4,//i == 4 is the smile
                            MOUTH_START_ANGLE, MOUTH_DEGREES_SHOWN);
            //Add text:
            canvas.drawString("Tee Hee.",
                      X_FACE0 + 50*i + FACE_DIAMETER, Y_FACE0 + 30*i);
        }
    }
```

图3-20 （续）

得到的applet

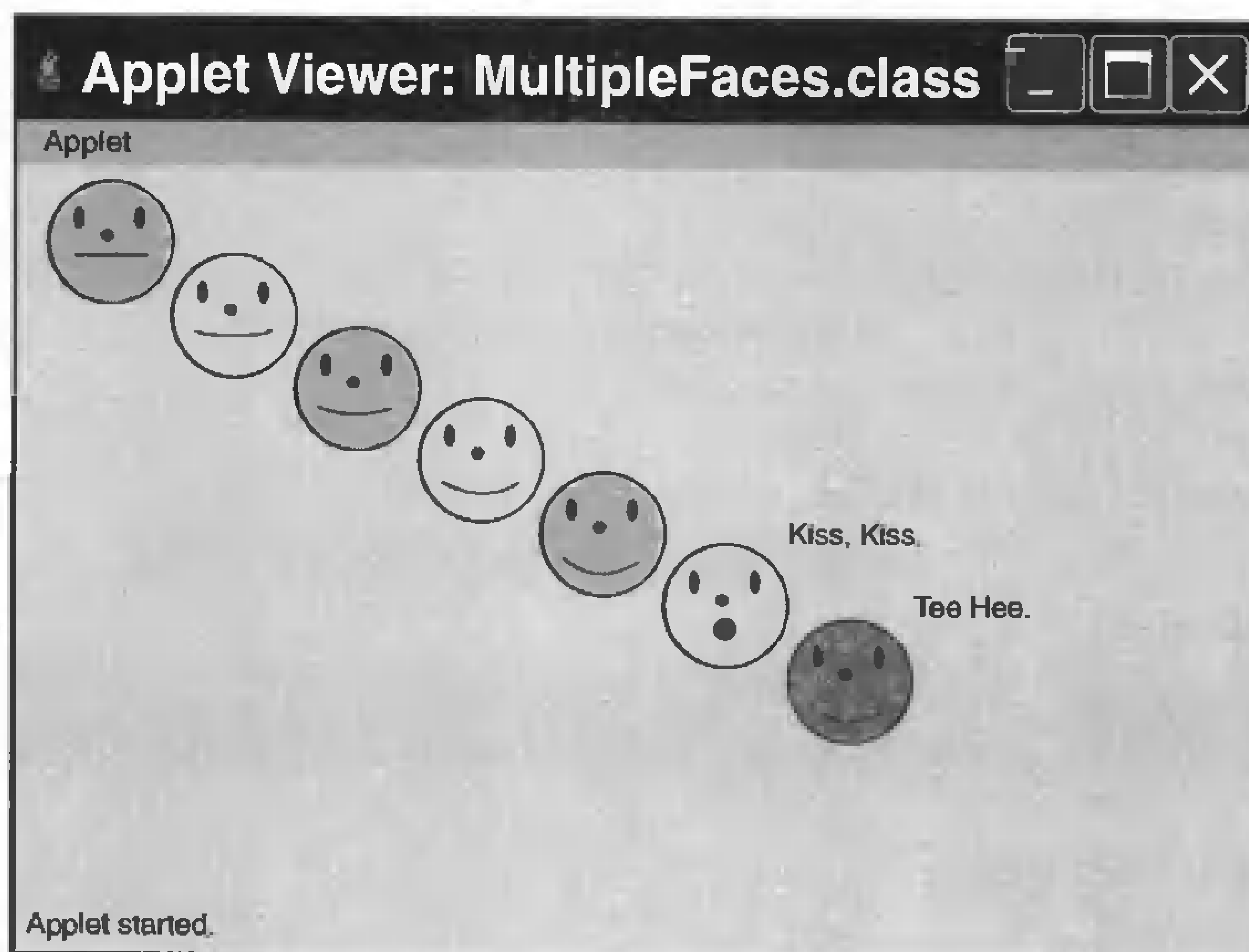


图3-20　（续）

除少量变化之外，前5张脸都是一样的，因此，可以用一个带有循环控制变量i的for循环给出。for循环的主体会画出一张随i值不同而稍微有些变化的脸孔。对每个i值来说，脸孔都是画在点（X_FACE0 + 50*i, Y_FACE0 + 30*i）上的。因此，当i为0时，第一张脸是画在点（X_FACE0, Y_FACE0）上的。对后续的每个i值来说，都会在屏幕上向右50像素，向下30像素的位置画另一张脸。X_FACE0和Y_FACE0是已定义常量。

只要i为偶数，下列代码就会向脸上添加黄色：

```
if (i%2 == 0) //if i is even
{//Make face yellow
    canvas.setColor(Color.YELLOW);
    canvas.fillOval(X_FACE0 + 50*i, Y_FACE0 + 30*i,
              FACE_DIAMETER, FACE_DIAMETER);
}
```

之后，用下列代码画出圆的黑色轮廓：

```
canvas.setColor(Color.BLACK);
canvas.drawOval(X_FACE0 + 50*i, Y_FACE0 + 30*i,
              FACE_DIAMETER, FACE_DIAMETER);
```

一定要注意，黄色的实心圆是在圆的黑色轮廓之前画的。我们希望黑色轮廓在黄色的实心圆之上，以显示出黑色轮廓。

循环体中接下来的几行画出了眼睛和鼻子，这些部分对每张脸都是一样的。

对每个i值来说，嘴都略有不同。嘴是由循环体末尾的下列代码行画出的：

```
//Draw mouth;
canvas.setColor(Color.RED);
canvas.drawArc(X_MOUTH0 + 50*i, Y_MOUTH0 + 30*i,
              MOUTH_WIDTH, MOUTH_HEIGHT0 + 3*i,
              MOUTH_START_ANGLE, MOUTH_DEGREES_SHOWN);
```

注意，除了嘴的高度随i加1而增加了3个像素之外，这些嘴都是一样的。i等于0时，嘴的高度为MOUTH_HEIGHT0，这是一个设置为0的已定义常量，因此第一张嘴是笔直穿过，没有弧度的。随着嘴高度的增加，曲线也更加明显，看起来也就更像笑容了。

最后两张脸孔是由循环体之后的代码给出的。但是，最后这两张脸孔中的代码都只对循环体作了少量修改。而且，与for循环生成脸孔的方法一样，最后两张脸孔的定位也是根据i来确定的。for循环终止时，i值被加了1，所以下一张脸孔也按次序正确定位了。在画最后一张脸孔之前，对i进行了递增操作，这样，最后一张脸孔也按脸孔序列正确定位了。

为了使倒数第二张脸上的嘴唇看起来是在亲吻的样子，把嘴唇画成了一个红色的实心圆。

最后一张脸是粉红色的，并给出了一个完整的笑容。将i设置为4就可以得到一个完整的笑容，因此，最后一张脸孔上嘴的高度就是按照i等于4来设置的。

最后两张脸孔用方法drawString添加了一些文字，将在3.5.2节进行解释。

3.5.2 drawString方法

方法drawString与画椭圆和长方形的方法类似，但drawString显示的是文字，而不是图。例如，下列代码会将字符串"Hello"写在从点（10,20）开始的地方：

```
canvas.drawString("Hello", 10, 20);
```

图3-20中的下列代码从x、y坐标为X_FACE0 + 50*i + FACE_DIAMETER和Y_FACE0 + 30*i的点开始，写下了字符串"Kiss, Kiss"：

```
//Add text
canvas.drawString("Kiss, Kiss.",
X_FACE0 + 50*i + FACE_DIAMETER, Y_FACE0 + 30*i);
```

快速参考：drawString方法

drawString方法会将实参*String*给出的文本写在applet的点（*X*，*Y*）上。（我们最终会说明，有时可以用一个标识符而不是canvas作为调用对象，但到目前为止，我们总是会使用canvas。）

语法：

```
canvas.drawString(String, X, Y);
```

举例：

```
canvas.drawString("Hello", 10, 20);
```

3.5.3 JOptionPane的确认窗口

本小节与本章其他图形编程内容无关。如果愿意，可以在学习本章其他图形编程内容之前学习本小节。如果没学过JOptionPane的有关内容，可以略过本小节。

有一种JOptionPane版本可以产生一个向用户提出Yes/No问题的窗口。窗口中包含指定的问题文本，以及两个标记为Yes和No的按钮。例如：

```
int answer =
    JOptionPane.showConfirmDialog(null, "End program?",
                   "Want to end?", JOptionPane.YES_NO_OPTION);
if (answer == JOptionPane.YES_OPTION)
    System.exit(0);
```

```
else if (answer == JOptionPane.NO_OPTION)
    System.out.println("One more time");
else
    System.out.println("This is impossible");
```

这段代码会生成图3-21中所示的窗口。如果用户点击Yes按钮，方法调用会返回对应于Yes按钮的int值，且窗口消失。由于返回的值为JOptionPane. YES_OPTION，多分支if-else语句就会调用System.exit(0)来终止程序。如果用户点击No按钮，方法调用会返回int值JOptionPane.NO_OPTION，多分支if-else语句会调用System.out.println将One more time写到屏幕上。这段代码被嵌入到文件JOptionPaneYesNoDemo.java的一个完整的程序之中，这个文件包含在本书的源代码中，本书的源代码可以在Web上找到。

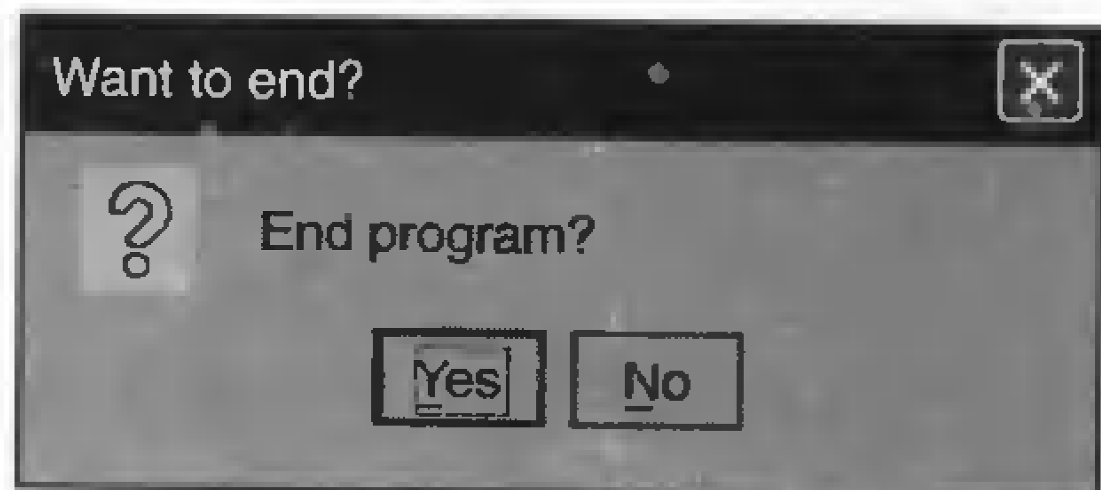


图3-21 是确认窗口

JOptionPane.showConfirmDialog返回的是一个int值，但你并不想把它当成一个int值，而是想把这个返回值当成对Yes/No问题的回答。为了使这种想法更易实现，JOptionPane类为这两个int值定义了名字。常量JOptionPane.YES_OPTION是点击Yes按钮时返回的int值，常量JOptionPane.NO_OPTION是点击No按钮时返回的int值。只是，JOptionPane.YES_OPTION和JOptionPane.NO_OPTION表示的是哪些int值呢？这并不重要，只是不要把它们当成int值。

目前我们学习的内容还不足以对实参列表进行完整的解释，但可以对大部分实参进行解释。例如

```
(null, "End program?",
        "Want to end?", JOptionPane.YES_NO_OPTION);
```

可以用任何其他字符串取代字符串"End program?"，那个字符串就是出现在带有Yes和No按钮的窗口中的字符串。当然，字符串通常都应该是一个答案为是或不是的问题。

可以用任何其他字符串取代字符串"Want to end?"，那个字符串就是作为窗口标题显示的字符串。

最后一个实参JOptionPane.YES_NO_OPTION说明你想生成一个带有Yes和No按钮的窗口。还有其他可能的选项，但在此不讨论。

第一个实参与将窗口放在屏幕上的什么位置有关，但我们学习的内容还不足以考虑其他可能的选择。因此，现在只要用null作为第一个实参就行了，null是默认值。

自测题

44. 为一个JOptionPane窗口编写一段代码，窗口会询问用户是否至少已经18岁了，如果用户说他至少18岁了，就将boolean变量adult设置为true，否则将其设置为false。代码中要包含变量adult的声明。

小　结

■ 从很多动作中选择一个来执行的语句被称为分支语句。if-else语句和switch语句都是分支语句。

- Java有两种形式的多路分支语句：switch语句和多分支if-else语句。
- 循环是一种编程结构，它会将一个动作重复数次。重复的部分称为循环体。循环体的每次重复被称为一次循环迭代。
- Java有3种类型的循环语句：while语句、do-while语句和for语句。
- 有一种可以用来结束输入循环的方法：在输入列表的末尾放一个标记值，并让循环来检测这个标记值。
- 最常见的循环错误种类为非预期无限循环和循环次数差一次错误。
- 跟踪一个变量就意味着每次变量被修改时都要将这个变量的值输出。这项功能可以通过专门的调试工具实现，也可以通过插入临时输出语句来实现。（有时并不需要将每次变化都输出，只需要有选择地输出其中一些变化就可以了。）
- 可以将布尔表达式的值存储在一个boolean类型的变量中。然后，就可以用这个变量来控制一条if-else语句或while语句了，也可以将其用在任何可以使用布尔表达式的地方。
- 可以向applet图画中添加颜色。
- 生成一个带有询问是与否问题的JOptionPane窗口。①

✓自测题答案

1.
```
if (goals > 10)
    System.out.println("Wow");
else
    System.out.println("Oh Well");
```

2.
```
if ((goals > 10) && (errors == 0))
    System.out.println("Wow");
else
    System.out.println("Oh Well");
```

3.
```
if (salary >= deductions)
{
    System.out.println("OK");
    net = salary - deductions;
}
else
{
    System.out.println("Crazy");
}
```

省略else部分的花括号也是可以的。

4.
```
if ((speed > 25) && (visibility < 20))
{
    speed = 25;
    System.out.println("Caution");
}
```

5.
```
if ((salary >= MIN_SALARY) || (bonus >= MIN_BONUS))
    System.out.println("OK");
else
    System.out.println("Too low");
```

6.
```
String upperWord = nextWord.toUpperCase();
if (upperWord.compareTo("N") < 0)
```

① 3.5节中介绍的内容。

```
    System.out.println("First half of the alphabet");
else
    System.out.println("Second half of the alphabet");
```

7. 如果它们都是int类型的，可以用x1==x2。如果它们都是String类型的，可以用x1.equals(x2)。

8. Time and tide wait for me.

9. Time and tide wait for no one.

10. Time and tide wait for every one.

11.
```
if (number > 10)
    System.out.println("High");
else if (number < 5)
    System.out.println("Low");
else
    System.out.println("So-so");
```

12. Till we meet again.

13.
```
Hello
Good-bye
```

14. Some kind of B.

15. Pie

16. Cookies

17. Diet time

18.
```
0
1
2
3
4
count after loop = 5
```

19. 是的，while循环的主体可以执行0次。
不行，do-while循环的主体必须至少执行一次。

20.
```
0
count after loop = 1
```

21.
```
Scanner keyboard = new Scanner(System.in);
int number;
{
    System.out.println("Enter a whole number:");
    number = keyboard.nextInt();
    System.out.println("You entered " + number);
}
while (number > 0)
{
    System.out.println("Enter a whole number:");
    number = keyboard.nextInt();
    System.out.println("You entered " + number);
}
System.out.println("number after loop = " + number);
```

22. 这是个无限循环。循环之后的println语句永远都不会执行。输出以下列形式开始：
```
0
-1
-2
-3
```

```
.
.
.
```

23.
```
1
2
3
4
```

24. 这个循环不会产生输出。第一次执行循环时，就不满足布尔表达式n>4，因此循环没有迭代循环体就终止了。

25.
```
4
3
2
1
```

26. 唯一的输出是

```
0
```

一定要注意那个添加到for循环第一行末尾的分号。

27.
```
0.0
0.5
1.0
1.5
2.0
2.5
```

28.
```
int n;
for (n = 1; n <= 5; n++)
    System.out.println(2*n);
```

29.
```
Hello
Hello
After the Loop.
```

30.
```
Hello
Hello
```

注意，没有输出"After the Loop."，因为程序终止了。

31.
```
One
Two
Three
After the Loop.
```

注意，break语句终止了switch语句，但并没有终止for循环。

32.
```
int time;
for (time = 1; time <= 4; time++)
    System.out.println("One more time.");
```

33.
```
int result = 1;
int count;
for (count = 1; count <= 5; count++)
result = 2*result;
```

34.
```
0
0
1
0
1
```

2

35. 可以用任何小于1.0的数作为标记值，但要避免任何由double值的近似特性带来的问题，选择的数字最好大大小于1.0。

```
double sum = 0, next;
System.out.println("Enter a list of numbers. All the");
System.out.println("numbers must be 1.0 or larger");
System.out.println("Place a zero at the end");
System.out.println("to mark the end of the list.");
Scanner keyboard =  new Scanner(System.in);
next = keyboard. nextDouble() ;
int count = 0;
while(next > 0.9)
             //next >=1.0 runs a risk of being inaccurate.
{
      sum = sum + next;
      count++;
      next = keyboard, nextDouble();
}
if(count > 0)
      System.out.println ("Average is " + (sum/count));
else
      System.out.println ("No numbers to average.");
```

36. 代码中包含的是

```
balance = balance + penalty;
balance = balance - deposit;
```

而它应该包含的是

```
balance = balance - penalty;
balance = balance + deposit;
```

即使用这种方法对其进行了修正之后，它仍然存在下列问题：如果penalty大于deposit，它就是个无限循环。在3.3.4节中对此进行了讨论。

37.

```
int n, sum = 0;
System.out.println("sum == " + sum);
for (n = 1; n < 10; n++)
{
     sum = sum + n;
     System.out.println("n == " + n);
     System.out.println("sum == " + sum);
}
System.out.println("1 + 2 + ··· + 9 + 10 == " + sum);
```

38. 布尔表达式应该是n<=10，而不是n<10。这是循环次数差一错误。

39. Positive

40. 产生的输出为

```
false
false
true
true
```

41. 产生的输出为

```
false
true
true
```

由于采用了短路求值，所以你不需要知道x的值。

42. 嘴看不见了，因为它会被黄色的实心圆盖住。脸上其他部分会和图3-18所示的一样。
43. 画出来的脸会和图3-18中的脸完全一样。
44. 不要求你给出一个完整的程序，但我们将答案嵌到一个完整的程序中了。

```
import javax. swing.*;

public class ExerciseJOptionPane
{
    public static void main (String [] args)
    {
        boolean adult = false;
        //Initialized to keep the compiler happy.

        int answer =
            JOptionPane. showConfirmDialog (null,
                        "Are you 18 years old or older?",
                        "Age Check" , JOptionPane.YES_NO_OPTION),
        if (answer == JOptionPane.YES_OPTION)
            adult = true;
        else if (answer == JOptionPane.NO_OPTION)
            adult = false;
        else
            System.out.println("Error");
        if (adult)
             JOptionPane. showMessageDialog(null, "You are old enough.");
        else
             JOptionPane. showMessageDialog(null, "Sorry, you must be 18.");
        System.exit(0);
    }
}
```

编程项目

1. 编写一个程序，将一行句子作为输入，并输出下列内容作为响应：如果句子是以问号标记'?'结束的，而且输入中包含了偶数个字符，就输出单词Yes；如果句子以问号标记'?'结束，且输入中包含了奇数个字符，就输出单词No；如果句子以感叹号'!'结束，输出单词Wow；在所有其他情况下，程序都会输出字符串You always say ，后面跟着用引号引起来的输入字符串。输入应该都在一行上。一定要注意，在最后一种情况下，输出一定要用引号将回送的输入字符串包围起来。在所有其他情况下，输出中都没有引号。程序中应该有一个循环，允许用户重复这个过程，直到用户表示他想结束这个程序为止。程序不需要查看输入以确定用户是否输入了一条合理的句子。
2. 编写一个程序，使用户可以将温度从摄氏度转换成华氏度，或者从华氏度转换成摄氏度。使用下列公式：

$$degreesC = 5(degreesF - 32)/9$$
$$degreesF = (9(degreesC)/5) + 32$$

提示：用户输入一个温度，以及用来表示摄氏的'C'（或'c'），或用来表示华氏的'F'（或'f'）；可以使用大写字母也可以使用小写字母，但如果输入的是除了'C'、'c'、'F'、'f'之外的任何其他

内容，就打印一条错误消息，并请用户重新输入一个合法的字符（大写或小写的'C'或'F'）。如果输入的是摄氏度，就将其转换成华氏度，如果输入的是华氏度，就将其转换成摄氏度，然后请用户按下Q或q键退出，或按下任意其他键重复循环，执行另一次会话。

3. 编写一个程序，读入一个非负整数序列，并输出最大整数、最小整数和所有整数的平均值。输入结束是由用户输入的一个负标记值来标识的。注意，查找最大、最小或平均值时不使用标记值。标记值只是一个结束标记。平均值应该是一个double类型的值，这样计算出来的平均值可以带有小数部分。

4. 编写一个程序，读入一个考试分数列表（0～100范围内的整数百分比），并输出总人数和每个字母等级类别中的人数（90~100 = A，80~89 = B，70~79 = C，60~69 = D，0~59 = F）。输入的结束是由一个负的分数作为标记值来识别的（负数只用来结束循环，因此在计算过程中不要使用它。）例如，如果输入为

```
98
87
86
85
85
78
73
72
72
72
70
66
63
50
-1
```

输出应该为

```
Total number of grades = 14
Number of A's = 1
Number of B's = 4
Number of C's = 6
Number of D's = 2
Number of F's = 1
```

5. 将编程项目3和4中的程序组合起来，读入考试分数（从0~100的整数百分比），并将下列统计信息打印出来：
   - 总分数。
   - 每个字母等级中的分数。
   - 每个字母等级中分数占总分数的百分比。
   - 数范围：最低分和最高分。
   - 平均分数。

   和以前一样，输入一个负的分数值作为标记值来终止数据输入，并将统计数据打印出来。

6. 编写一个程序，将银行账户余额和利率作为输入，并输出10年内的账户值。输出应该将用3种不同的复利计算方式得到的账户值都显示出来：按年的、按月的和按日的。按年计算复利时，利率在每年结束时添加一次。按月计算复利时，利率每年添加12次。按日计算复利时，利率每年添加365次。不用

考虑闰年的问题，假设所有年份都有365天。对年利率来说，可以假设利率会在存款日期之后正好一年的时候发放，即无需考虑要在一年中某个特殊的日子来发放利率，比如12月31日。类似的，可以假设月利率是在存款之后正好一个月的时候发放。因为账户还会得到利息的利息，所以，利息发放越频繁，账户的余额就应该越高。一定要根据利率的时间周期来调整利率。如果年利率为5%，那么按月发放利率时，就要使用(5/12)%。按日计算利率时，就要使用(5/365)%。用循环来实现为每个时间周期添加利息的计算过程（即不要用某些算术公式来计算）。程序应该有一个外围循环，使用户可以为新的账户余额和利率重复这样的计算过程。重复进行计算，直到用户表示要终止程序为止。

7. 对第2章的编程项目9进行修改，以包含输入检测。只有在输入有效价格（不少于25美分，不多于100美分，是一个面值为5美分倍数的整数）时才打印零钱方案。否则，为下列每种无效输入分别打印错误信息：低于25美分的费用，不是5的整数倍的费用，以及大于1美元的费用。

8. 编写一个程序，请用户输入要打印的三角形的尺寸（1~50的整数），然后通过打印一系列包含星号的行将三角形打印出来。第一行上只有一个星号，第二行上有两个，以此类推，每行都比前一行多一颗星，直到达到用户输入的数字为止。在下一行上少打印一颗星，接下来的每一行都将星号的数量减1，直到只打印出一个星号为止。提示：使用嵌套for循环；外围的循环控制要打印的行数，内层的循环控制要在一行上打印的星号数量。例如，如果用户输入5，输出将为

```
*
**
***
****
*****
****
***
**
*
```

9. （要完成这个项目，需要学习3.5节的内容。）编写一个applet程序，用来显示一系列有胳膊、腿，当然还有头的人的图像。用笑脸作头，用椭圆作身体、胳膊和腿。画出一系列如图3-20中所示一个接一个地出现的形象。使人物形象呈现奔跑的姿势。改变每个后继形象中人物脸孔的颜色，从白到粉红到红到黄到绿。逐渐地改变笑脸上嘴唇的形状，从第一个人的微笑改成最后一个人的皱眉。用一条switch语句来选择颜色。将switch语句嵌在一个循环之中。

第4章 定义类与方法

class 名词 1.a.集合、集团、组，或成员具有（或认为它们具有）至少一个相同属性的结构；种类；分类……

——《美国传统英语词典》（第3版）

对象是由一个类类型的变量命名的。对象中包含数据，也可以执行动作。它们执行的动作称为方法。前面已经使用过一些对象了。String类型就是一个类，String类型的值就是对象。例如，如果name是一个String类型的对象，就可以用方法length来确定这个字符串的长度。表达式name.length()返回的值就是字符串的长度。本章将介绍如何定义自己的简单类，以及如何使用对象和这些类中的方法。

目标

- 熟悉类的概念，以及用对象对类进行实例化的概念。
- 学会如何在Java中定义类。
- 学会在Java中定义并使用方法（对象的动作）。
- 学会在Java中创建对象。
- 弄清楚在Java中参数是如何工作的。
- 学习信息隐藏与封装的相关知识。
- 熟悉引用的概念，以便理解类变量和类参数的概念。
- 选读，继续在4.4节中对applet进行讨论。

预备知识

在阅读本章之前，需要熟悉第2章和第3章的内容。

4.1 类和方法的定义

Java程序是由从各种类中导出的、互相交互的对象组成的。在开始讨论如何在Java中定义并使用类和对象的细节之前，对类和对象有个大致的了解将是很有帮助的。

可以用**对象**来表示现实世界中的各种对象，比如汽车、房屋、雇员记录——几乎所有你想表示的东西。**类**是一类对象的定义。它就像一个用来构建某些特定对象的轮廓或计划一样。例如，图4-1描述了一个名为Automobile的类。这个类对汽车是什么、能做些什么进行了总体性的描述。对象就是特定的汽车。图中显示了3个Automobile对象。我们称满足了Automobile类定义的对象对Automobile类进行了**实例化**（instantiate）。因此，对象是指单个

汽车，而Automobile类则描述了汽车是什么及汽车能干什么。当然，这是一个相当简化的Automobile类，但它说明了类的基本思想。下面来看一些细节。

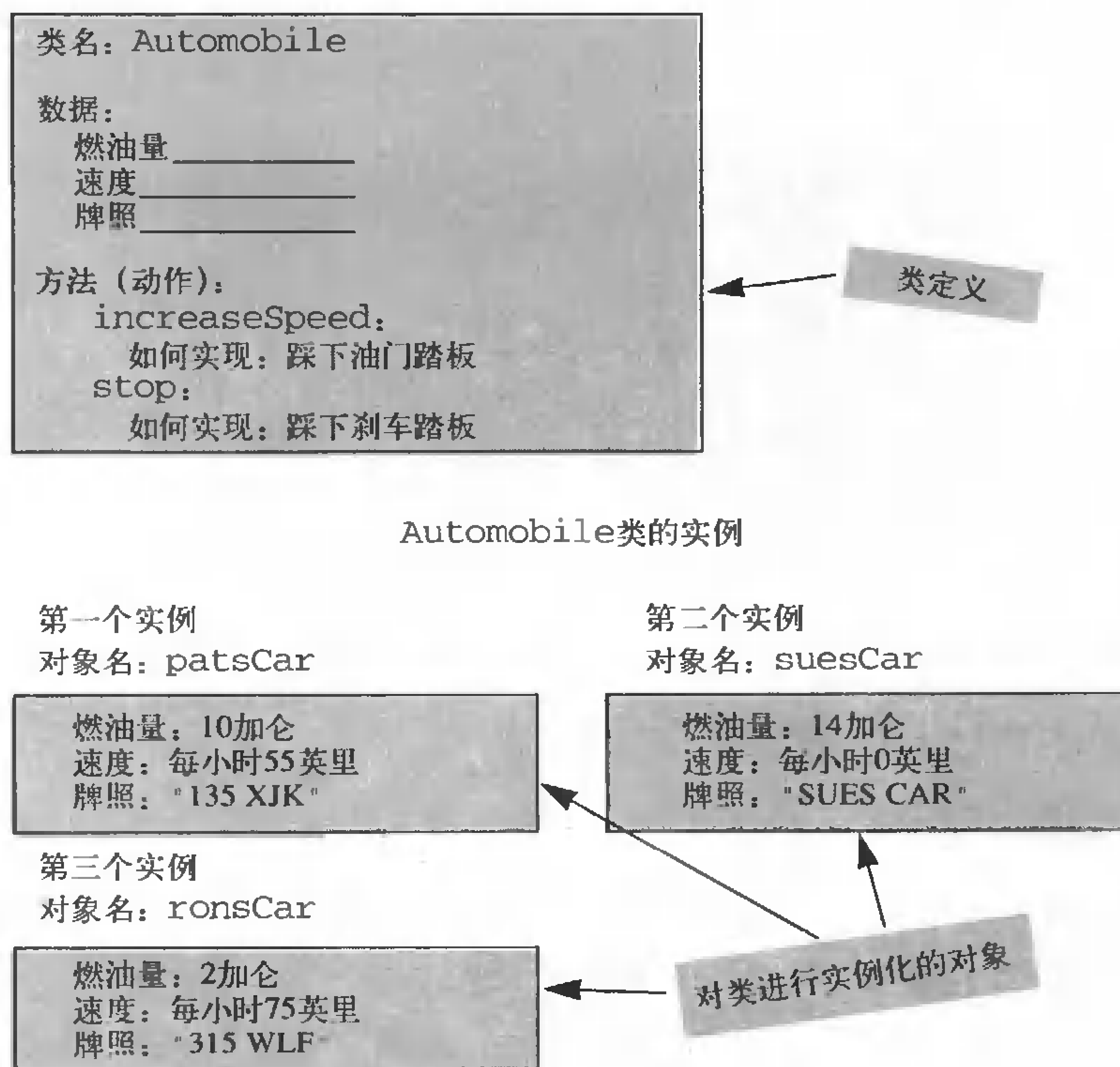


图4-1 类的轮廓

类指定了其对象所拥有的数据类型。Automobile类的定义表明一个Automobile对象有3份数据：一个数字用来说明燃料箱里还有多少加仑燃料，另一个数字用来说明汽车跑得有多快，还有一个用来显示牌照上所写内容的字符串。类定义中没有数据（也就是说，没有数字和字符串）。单个的对象中拥有数据，但类说明了它们能拥有什么类型的数据。

类还指定了对象可以执行哪些动作，以及它们是如何完成这些动作的。Automobile类中指定了两个动作：increaseSpeed和stop。因此，在使用了Automobile类的程序中，一个Automobile对象所能执行的行动只有increaseSpeed和stop。这些动作称为**方法**。Automobile类的所有对象都拥有相同的方法。任何一个类的所有对象都拥有同样的方法。正如从Automobile类的示例中可以看到的，方法的定义（动作是如何执行的）是在类定义中给出的。但是，方法的动作是由对象执行的。

图4-1中的表示法有点儿累赘，因此，程序员们通常会用一种更简单的图形表示法来总结类的一些主要特征。图4-2显示的这种表示法被称为**UML类图**（UML class diagram）或简称**类图**（class diagram）。（UML即统一建模语言（Universal Modeling Language）。图4-2中描述的类与图4-1中描述的类是一样的。本章稍后将对图4-2中的新注解进行解释。

还有另外几个与类和对类进行实例化的对象有关的事项需要注意。每个对象都有一个名字。在图4-1中，对象的名字为patsCar、suesCar和ronsCar。除此之外还要注意，类是一种数据类型。在Java程序中，patsCar、suesCar和ronsCar这些对象名都是Autombile类型的变量。

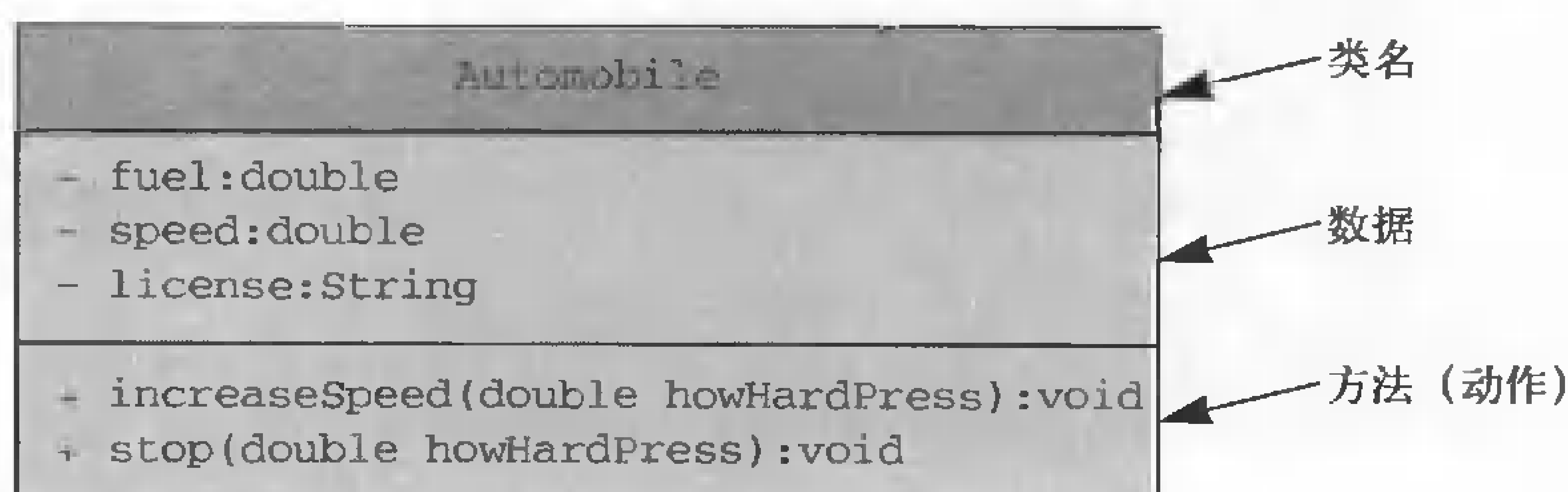


图4-2　用UML类图描述的类轮廓

在通过定义一个简单的类来深入了解Java代码的具体细节之前，我们要说明如何将类存储到文件中，以及如何对其进行编译。

4.1.1　类文件及独立编译

无论你使用的是本书中的类，还是自己编写的类，都需要了解一些关于如何将Java类定义存储到文件中的基本细节。每个Java类定义都应该是一个单独的文件[①]，文件名应该与类名相同，而且文件名应该以.java结尾。因此，如果你为一个名为Automobile的类编写了一个定义，就应该将其存放在一个名为Automobile.java的文件中。如果你为一个名为MyClass的类编写了一个定义，这个定义就应该在一个名为MyClass.java的文件中。

在编写一个程序来使用一个Java类之前，可以对这个Java类进行编译。编译好的类字节码存储在一个同名、但以.class而不是.java结尾的文件中。因此，对文件Automobile.java的编译会创建一个名为Automobile.class的文件。稍后，编译一个在main部分中使用了Automobile类的程序文件时，就不需要再重新编译Automobile的类定义了。这种命名要求适用于整个程序，也适用于类。注意，每个带有main部分的程序，在文件起始处都有一个类名；这就是包含这个程序的文件应该使用的名字。例如，图4-4中的程序应该存放在一个名为SpeciesFirstTryDemo.java的文件中。只要程序中用到的所有类都和程序文件位于同一个目录中，就不用担心目录问题。第5章会介绍如何把文件放到多个目录中。

4.1.2　实例变量

图4-3包含了一个简单的类定义。我们对这个类进行了简化，使这个例子更容易理解。本章稍后，将用更好的形式来描述同一个例子。但这个例子中包含了一个类定义必需的所有内容。

① 这条规则是有些例外情况的，但我们很少会碰到，所以不需要去关心。

```
import java.util.*;

public class SpeciesFirstTry
{
    public String name;
    public int population;
    public double growthRate;

    public void readInput()
    {
        Scanner keyboard = new Scanner(System.in);
        System.out.println("What is the species' name?");
        name = keyboard.nextLine();

        System.out.println("What is the population of the species?");
        population = keyboard.nextInt();
        while (population < 0)
        {
            System.out.println("Population cannot be negative.");
            System.out.println("Reenter population:");
            population = keyboard.nextInt();
        }

        System.out.println(
                    "Enter growth rate (percent increase per year):");
        growthRate = keyboard.nextDouble();
    }

    public void writeOutput()
    {
        System.out.println("Name = " + name);
        System.out.println("Population = " + population);
        System.out.println("Growth rate = " + growthRate + "%");
    }

    public int populationIn10()
    {
        double populationAmount = population;
        int count = 10;
        while ((count > 0) && (populationAmount > 0))
        {
            populationAmount = (populationAmount +
                          (growthRate/100) * populationAmount);
            count--;
        }
        if (populationAmount > 0)
            return (int)populationAmount;
        else
            return 0;
    }
}
```

本章稍后将给出这个类的更好版本。

本章稍后将介绍应该用修饰符private代替public。

如第2章所述，(int)是一种强制类型转换。

图4-3 类定义

这个类的类名为SpeciesFirstTry，是设计用来表示濒危物种记录的。（称其为FirstTry是因为稍后会给出这个类的改进版本。）这个类的每个对象都有3份数据：名字、种群规模和增长率。这些对象都有3个方法：readInput、writeOutput和populationIn10。因为数据项和方法都是属于对象的，所以有时会将它们称为对象的**成员**（member）；有时也会将它们称为**域**（field）。但是，我们会将数据项称为**实例变量**（instance variable），并将方法

称为**方法**（method）。先来讨论数据项（即实例变量）。

下面这3行位于类定义起始处的代码定义了3个实例变量（3个数据成员）：

```
public String name;
public int population;
public double growthRate;
```

public只是说明这些实例变量的使用是没有限制的。每行代码都声明了一个实例变量名。可以把类的对象当成一个包含了一些实例变量的复杂项。因此，也可以把实例变量当成每个类对象内部的一个较小变量。在这个例子中，实例变量称为name、population和growthRate。类的每个对象都拥有这3个实例变量。图4-4中的程序显示了这个类定义的用法。下面来看看它是如何处理这些实例变量的。

```
public class SpeciesFirstTryDemo
{
    public static void main(String[] args)
    {
        SpeciesFirstTry speciesOfTheMonth = new SpeciesFirstTry();
        int futurePopulation;

        System.out.println("Enter data on the Species of the Month:");
        speciesOfTheMonth.readInput();
        speciesOfTheMonth.writeOutput();

        futurePopulation = speciesOfTheMonth.populationIn10();
        System.out.println("In ten years the population will be "
                                        + futurePopulation);
        speciesOfTheMonth.name = "Klingon ox";
        speciesOfTheMonth.population = 10;
        speciesOfTheMonth.growthRate = 15;
        System.out.println("The new Species of the Month:");
        speciesOfTheMonth.writeOutput();
        System.out.println("In ten years the population will be "
                         + speciesOfTheMonth.populationIn10());
    }
}
```

屏幕对话示例

```
Enter data on the Species of the Month:
What is the species' name?
Ferengie fur ball
What is the population of the species?
1000
Enter growth rate (percent increase per year):
-20.5
Name = Ferengie fur ball
Population = 1000
Growth rate = -20.5%
In ten years the population will be 100
The new Species of the Month:
Name = Klingon ox
Population = 10
Growth rate = 15.0%
In ten years the population will be 40
```

图4-4　类和方法的使用

图4-4中的下面这行代码创建了一个SpeciesFirstTry类型的对象，并将名字speciesOfTheMonth赋予了这个对象：

```
SpeciesFirstTry speciesOfTheMonth = new SpeciesFirstTry();
```

与所有SpeciesFirstTry类型的对象一样，对象speciesOfTheMonth有3个名为name、population和growthRate的实例变量。引用实例变量的方法如下所示：写下对象名，后面跟一个点，然后跟实例变量的名字。例如：

```
speciesOfTheMonth.name
```

表示的是对象speciesOfTheMonth的实例变量name。再来看看定义实例变量的这3行代码：

```
public String name;
public int population;
public double growthRate;
```

注意，每个实例变量都有一个类型。例如，实例变量name是String类型的，因此实例变量speciesOfTheMonth.name就是一个String类型的变量，可以用于任何能够使用String类型变量的地方。例如，下列所有语句都是有效的Java表达式：

```
speciesOfTheMonth.name = "Klingon ox.";
System.out.println("Save the " + speciesOfTheMonth.name);
String niceName = speciesOfTheMonth.name;
```

每个SpeciesFirstTry类型的对象都有3个自己的实例变量。例如，假设程序中还包含下列语句：

```
SpeciesFirstTry speciesOfLastMonth = new SpeciesFirstTry();
```

那么，speciesOfTheMonth.name和speciesOfLastMonth.name将是两个不同的实例变量，可能具有不同的字符串值。

常见问题：为什么要用new？

将new用于下面这样的表达式时，可以认为它是在创建对象的实例变量。

```
SpeciesFirstTry speciesOfLastMonth = new SpeciesFirstTry();
```

像speciesOfLastMonth这样的类类型对象，其内部可以包含更小的变量，即对象的实例变量。new将这些实例变量放在对象内部。4.3节将对new的这种用法给出更完整的解释。

4.1.3 方法的使用

使用一个方法时，称为**调用**了这个方法。前面已经调用过方法了。比如，你的程序用Scanner类的对象调用过方法nextInt()。如下列语句所示，你还用对象System.out调用过方法println：

```
System.out.println("Enter data on the Species of the Month:");
```

方法有两种类型：返回单个值的方法，执行某些动作而不返回单个值的方法。方法nextInt就是返回单个值的方法。它返回了一个int类型的值。方法println是执行某些动作而不返回单个值的方法。这两种不同类型方法使用的方式略有不同。

记住：两种类型的方法

有两种类型的方法：返回单个值的方法；执行某些动作而不返回单个值的方法。执行某些动作而不返回值的方法被称为void方法。

先以方法readLineInt为例，讨论如何调用一个会返回单个值的方法。假设程序中有下列声明：

```
int next;
```

假定keyboard是Scanner类的对象，下面是通过对象keyboard调用方法nextInt的示例：

```
next = keyboard.nextInt();
```

（如果想在完整的程序上下文中看看这个调用过程，可以参考图3-17。）下面进一步查看这个方法调用的细节。

在类中定义的方法通常都是用那个类的对象调用的。这个对象被称为**调用对象**，这是在书写对象调用时要给出的第一项内容。可以这样来调用一个方法：书写调用对象名（比如keyboard），后面跟一个点，然后跟对象名（比如nextInt），最后跟一对圆括号，圆括号中可能包含（也可能不包含）需要传递给方法的信息。如果是要返回单个值的方法，比如方法nextInt，就可以在任何可以使用该方法返回值类型的地方使用这个方法调用。方法nextInt返回了一个int类型的值，因此，可以将方法调用

```
keyboard.nextInt()
```

用于任何可以使用int类型值的地方。例如，可以像下面这样，将6这样的int类型值用于一条赋值语句中：

```
next = 6;
```

因此，也可以像下面这样，用同样的方法来使用方法调用keyboard.nextInt()：

```
next = keyboard.nextInt()
```

调用一个返回单个值的方法时，就好像方法调用被返回值取代了一样。因此，如果keyboard.nextInt()返回的值为3，那么，赋值语句

```
next = keyboard.nextInt();
```

产生的效果就和

```
next = 3;
```

一样。

执行某些动作而不返回单个值的方法也是类似的，只是它们是用于包含一些Java语句，而不是用来产生返回值的。例如，图4-4程序中的下列语句包含了对对象System.out进行的println方法调用：

```
System.out.println("Enter data on the Species of the Month:");
```

这个方法调用将字符串"Enter data on the Species of the Month:"写到屏幕。图4-4中用到的SpeciesFirstTry类的方法writeOutput与之类似，只是不需要在圆括号中包含一些内容以告知writeOutput要输出些什么。方法writeOutput是从它的调用对象获得要发送到屏幕的信息的。

例如，图4-4中的程序（在完成其他一些任务之后）会用下列3条赋值语句来设置对象speciesOfTheMonth中实例变量的值：

```
speciesOfTheMonth.name = "Klingon ox";
speciesOfTheMonth.population = 10;
speciesOfTheMonth.growthRate = 15;
```

然后，程序会用下列语句将这些值输出：

```
System.out.println("The new Species of the Month:");
speciesOfTheMonth.writeOutput();
```

上面两行代码中的第二行包含了通过调用对象speciesOfTheMonth对方法writeOutput的调用。这次调用会产生下列输出：

```
Name = Klingon ox
Population = 10
Growth rate = 15.0%
```

要调用一个对象的方法，需要写下调用对象名（如speciesOfTheMonth），后面跟一个点，方法名（如writeOutput），最后跟一对圆括号，圆括号中可能会包含一些要传递给方法的信息。如果像这个例子一样，调用的是会执行某些动作而不是返回单个值的方法，就要在方法调用后面放一个分号，使其成为一条Java语句。比如，下面就是一条对speciesOfTheMonth对象的writeOutput方法的调用：

```
speciesOfTheMonth.writeOutput();
```

这样，方法就会执行方法定义中指定的任意动作了。

快速参考：方法调用（调用一个方法）

在调用对象后面写一个点，然后写上方法的名字，最后写上一对圆括号，圆括号中可能有（也可能没有）要传递给方法的信息，这样就可以调用一个方法了。

如果方法调用会返回一个值，就可以在任意一个可以使用方法返回值类型的地方使用这个方法调用了。例如，下列代码中包含了通过调用对象speciesOfTheMonth调用的方法populationIn10：

```
futurePopulation = speciesOfTheMonth.populationIn10();
```

如果方法调用执行了某些动作而没有返回单个值，就要在方法调用后面放一个分号，生成一条Java语句。（这些执行动作的方法被称为void方法。）例如，下面是通过调用对象speciesOfTheMonth 对void方法readInput进行的调用：

```
speciesOfTheMonth.readInput();
```

这个方法调用会使方法开始执行方法定义中指定的任意动作。

4.1.4 void方法定义

下列方法调用来自图4-4：

```
speciesOfTheMonth.writeOutput();
```

下面通过方法writeOutput的定义看看方法定义是怎么写。定义是在图4-3中给出的，如下所示：

```
public void writeOutput()        头部
{
    System.out.println("Name = " + name);
    System.out.println("Population = " + population);        主体
    System.out.println("Growth rate = " + growthRate + "%");
}
```

所有的方法定义都是属于某个类的，而且所有方法的定义都是在它们所属的类的定义中给出的。由图4-3可以发现这个方法定义是位于SpeciesFirstTry类定义内部的，这就意味着方法只能通过SpeciesFirstTry类的对象来使用。

不返回值的方法的定义是以关键字public void开始的，后面跟着方法名和一对圆括号。Public指明对方法的使用没有特别限制。有时可以用其他修饰符来取代public，以限制方法的使用。void是个很不好的选择，但Java和其他语言都在用它。void说明方法不会返回值。圆括号会将方法所需的任意额外信息括起来。在这个例子中，不需要额外的信息，所以圆括号中什么也没有。稍后在本章中会看到一些例子，其中显示了在其他方法定义的圆括号中可能会出现的内容。这是方法定义的第一部分，称为方法的**头部**。通常头部都是写在单独一行上的；但是，如果写成一行太长，也可以把它分成两行（或多行）。由于在方法头部使用了单词void，这些不返回值的方法被称为**void方法**。

跟在头部后面的是方法定义的**主体**（body），主体部分用来完成方法的定义。方法定义的主体包围在花括号{}之间。可以在花括号之间放置任何能放在程序main部分的语句或声明。除了类的实例变量之外，在方法定义中使用的任何变量都应该在相应方法定义内声明。

调用void方法时，就像方法调用被方法定义的主体取代了一样，会执行主体中的语句或声明。这个替换过程有些微妙之处，但对现在要看的这个简单示例来说，这个过程就像在字面上用方法定义的主体取代了方法调用一样。最终，还是要学着把方法定义看作定义了一个要执行的动作，而不是一系列用来替代方法调用的语句，但这种替代思想是正确的，而且是理解方法调用的好方式。

例如，下列方法调用出现在图4-4所示的程序中：

```
speciesOfTheMonth.writeOutput();
```

执行这个方法调用时，就好像方法调用的这行代码被方法writeOutput的方法定义主体取代了一样。在这个例子中，前面的方法调用就好像被下列代码取代了一样：

```
{
     System.out.println("Name = " + name);
     System.out.println("Population = " + population);
     System.out.println("Growth rate = " + growthRate + "%");
}
```

实例变量名（name、population和growthRate）是指调用对象的实例变量；在这个例子中，它们是指对象speciesOfTheMonth的实例变量。更精确地说，这个方法调用等价于下列代码：

```
{
     System.out.println("Name = " + speciesOfTheMonth.name);
     System.out.println("Population = "
                        + speciesOfTheMonth.population);
     System.out.println("Growth rate = "
                        + speciesOfTheMonth.growthRate + "%");
}
```

非常具体地来说，如果speciesOfTheMonth.name的值为"Klingon ox"，speciesOfTheMonth.population的值为10，speciesOfTheMonth.growthRate的值为15，则方法调用

```
speciesOfTheMonth.writeOutput();
```

就会将下列内容写到计算机屏幕：

```
Name = Klingon ox
Population = 10
Growth rate = 15.0%
```

从图4-4中的程序会发现它看起来就像一个没有实例变量，只有一个名为main的方法的类定义一样。main确实是一个方法。程序只是个拥有一个main方法的类。到目前为止，我们编

写的所有程序都没有实例变量，除了方法main之外也没有其他方法。但实际上程序是可以有其他方法，也可以有实例变量的。运行一个程序时，只是调用了一个名为main的void方法。当然，这是一种特殊类型的方法调用。现在，static和String[] args这些附加单词暂时还不解释。只要把它们加上就行了，最终我们会对所有这些单词进行解释的。

4.1.5 返回值的方法

会返回单个值的方法的定义方式与void方法的定义方式基本相同，只是有一个地方更复杂一些——要指定返回的值。下面看看SpeciesFirstTry类的方法populationIn10。在图4-4程序的下列行中使用了这个方法：

```
futurePopulation = speciesOfTheMonth.populationIn10();
```

这条语句将变量futurePopulation的值设置为方法调用

```
speciesOfTheMonth.populationIn10();
```

返回的值。

方法populationIn10的定义告诉了计算机如何计算返回值，下面是图4-3中这个方法的定义：

```
public int populationIn10()
{
    double populationAmount = population;
    int count = 10;
    while ((count > 0) && (populationAmount > 0))
    {
        populationAmount = (populationAmount +
                           (growthRate/100) * populationAmount);
        count--;
    }
    if (populationAmount > 0)
        return (int) populationAmount;
    else
        return 0;
}
```

与void方法定义一样，会返回值的方法的定义也可以分成两个部分：方法头部和方法主体。下面是方法populationIn10的头部：

```
public int populationIn10()
```

返回值的方法的头部与void方法头部的描述方式基本相同。唯一的区别是，返回值的方法用一个类型名取代了关键字void。头部以关键字public开头，后面跟着一个类型名（而不是单词void），后面再跟上方法名和一对圆括号。圆括号中包含了对方法所需的所有额外信息的描述。在这个例子中，不需要额外的信息，因此圆括号中是空的。关键字public指明方法的使用没有什么特别限制。稍后在本章中你会看到，可以用其他修饰符取代单词public，以限制方法的使用。这个方法中一个很重要的新要素，就是方法头部类型名的使用——在这个例子中就是int。

返回值的方法的头部包含了一个类型名。类型名就是返回值的类型。每个方法都只能返回一种类型的值。在不同情况下，方法可以返回不同的值，但这些值必须都是方法头部指定类型的值。

返回值的方法定义的主体与void方法定义的主体很像，只是它必须在一个或多个地方包含下列语句：

```
return Expression;
```

这条语句被称为**return语句**。*Expression*可以是任何能够生成一个方法定义头部指定类型值的表达式。这条语句说明方法返回的值就是这个表达式的值。例如，方法populationIn10的定义中包含了两条return语句：

```
return (int)populationAmount;
```

以及

```
return 0;
```

调用一个返回单个值的方法时，就会执行方法定义主体中的语句。例如，来自图4-4的方法调用：

```
futurePopulation = speciesOfTheMonth.populationIn10();
```

执行这条赋值语句时，就会执行populationIn10的方法定义主体。这个方法定义主体如下所示：

```
{
    double populationAmount = population;
    int count = 10;
    while ((count > 0) && populationAmount > 0))
    {
        populationAmount = (populationAmount +
                           (growthRate/100)* populationAmount);
        count--;
    }
    if (populationAmount > 0)
        return (int)populationAmount;
    else
        return 0;
}
```

如第2章所述，(int)是强制类型转换。

实例变量population是指调用对象的实例变量，在这个例子中，调用对象就是speciesOfTheMonth。将population的值复制到变量populationAmount中去，然后执行while循环。循环的每次迭代都会在populationAmount的值上加上一年内种群的数量变化，循环会迭代10次。因此，当while循环结束时，populationAmount的值就是10年内的计划种群规模。这时，populationAmount的值就是希望方法返回的值。现在，我们假设那个数字是正的（也就是说，那个物种没有灭绝）。在这种情形下，会执行下列return语句，这条语句表明值(int)populationAmount就是方法调用计算出来的（就是它返回的）值：

```
return (int)populationAmount;
```

(int)是一种强制类型转换，将double类型的值转换成一个int类型的值，这样动物的数量就不会有分数了。这就像方法调用被(int)populationAmount取代了一样。在这个例子中，方法调用出现在下列赋值语句中：

```
futurePopulation = speciesOfTheMonth.populationIn10();
```

因此，变量futurePopulation就被设置成speciesOfTheMonth.populationIn10() 的值。

如果populationAmount碰巧为零或负数，就会执行下列return语句：

```
return 0;
```

这是一个很小的细节，它可以确保预测种群规模不会为负数。毕竟，在现实世界中，一旦一个种群中的个体数为零了，那么这个种群数量就会保持为零，而不会变成负的。

执行return语句时，这条语句就决定了方法返回的值，也会终止方法的调用。如果return语句后面还有语句，那么这些语句都不会被执行。

返回值的方法还可以执行其他一些动作，如从键盘读入一个值，但它必须要返回一个值。

记住：命名方法

Java允许你用任何合法的（非关键字）标识符作为方法名。但是，如果选择一些清晰、有意义的名字，会使代码更易读。为方法命名需要遵循的一条很好的原则是：用动词来命名void方法，用名词来命名返回单个值的方法。这是因为void方法像动词一样命名了一个动作。另一方面，返回值的方法可以像一个值一样使用，而值是一件事物，名词则是用来标明事物的。

命名类和方法时遵循的常规约定是：所有的方法名都以小写字母开头，而所有的类名都以大写字母开头。

常见问题：什么是函数？

在其他一些编程语言中，返回值的方法称为函数，而一个会返回值的方法确实与函数的数学概念相对应。但是，在Java中它们称为（会返回值的）方法，而不是函数。

■ Java提示：void方法中return的用法

void方法不返回值，因此不需要任何return语句。但有一类return语句有时会在void方法用到。void方法中的return语句具有下列形式：

```
return;
```

除了（由于没有返回值）在这条语句中没有包含任何返回值的表达式之外，这条return语句和你见过的其他return语句是一样的。执行这条return语句时，void方法的调用终止。可以用这条语句提前终止一个方法调用，比如方法发现了某些类型的错误时。例如，可以将下列方法添加到SpeciesFirstTry类中去：

```
public void showLandPortion()
{
    if (population == 0
    {
        System.out.println("Population is zero.");
        return;//Ends here to avoid division by zero.
    }
    double fraction;
    fraction = 6.0/population;
    System.out.println("If the population were spread");
    System.out.println("over 6 continents, then each");
    System.out.println("individual would have a fraction of");
    System.out.println("its continent equal to" + fraction);
}
```

如果方法的其余部分会包含被零除的情况，方法就会以一条return语句作为结束。一般不太可能会有这样的方法，但这个例子确实说明了问题。

快速参考：方法定义

每个方法都属于某个类。方法的定义是在它所属类的定义中给出的。方法定义最常见的两种形式如下所示。

void方法定义：

```
public void Method_Name (Parameters)
}
    Statement_1
    Statement_2
    ...
    Statement_Last
}
```

(到目前为止我们还未介绍过*Parameter*（形参），不过很快就会介绍。如果没有*Parameter*，圆括号中就是空的。)

举例：

```
public void writeOutput()
{
    System.out.println("Name = " + name);
    System.out.println("Population = " + population);
    System.out.println("Growth rate = " + growthRate + "%");
}
```

返回值的方法的定义：

```
public Type_Returned Method_Name (Parameters)
{
<语句列表，其中至少包含了一条return语句。>
}
```

举例（可以将这段代码添加到图4-3的类中）：

```
public int halfThePopulation()
{
    return (population/2);
}
```

快速参考：return语句

每个返回单个值的方法的定义中都必须有一条或多条return语句。return语句用来说明方法返回的值，并终止程序的调用。

语法：

```
return Expression
```

举例：

```
public int halfThePopulation()
{
    return (population/2);
}
```

void方法中不一定要有return语句，但如果想在代码结束之前终止方法调用，它也可以使用return语句。void方法中return语句的形式为

```
return;
```

4.1.6 this参数

图4-3中有SpeciesFirstTry类的定义，图4-4中使用了这个类的程序。注意，根据实例变量位于类定义内部还是类定义之外，实例变量的写法是有所不同的。在类定义之外，可以通过给出类的对象名，后面跟一个点和实例变量名来表示一个实例变量，就像下列出现在图4-4中用来引用实例变量name的方法：

```
speciesOfTheMonth.name = "Klingon ox";
```

但在同一个类的方法定义内部引用实例变量时，只用实例变量名就行了，不需要任何对象名或点。比如，下列行出现在图4-3中SpeciesFirstTry类的readInput方法定义中：

```
name = keyboard.nextLine();
```

包括这个name在内的每个实例变量都是某个对象的实例变量。在这种情况下，我们认为对象是存在的，但通常会省略它的名字。这个对象有个有点儿不太寻常的名字——this。尽管通常会省略this（但我们认为它是存在的），但如果你想加的话，也可以把它加上。例如，图4-3的readInput方法定义中对实例变量name的赋值语句与下列语句是等效的：

```
this.name = keyboard.nextLine();
```

另外一个例子是，下列代码重新编写了方法writeOutput的方法定义。这个定义等效于图4-3中的版本：

```
public void writeOutput()
{
    System.out.println("Name = " + this.name);
    System.out.println("Population = " + this.population);
    System.out.println("Growth rate = " + this.growthRate + "%");
}
```

关键字this表示调用对象。比如，对图4-4中的方法调用来说：

```
speciesOfTheMonth.writeOutput();
```

调用对象为speciesOfTheMonth。因此这个对方法writeOutput的调用就等效于下列语句：

```
{
    System.out.println("Name = " + speciesOfTheMonth.name);
    System.out.println("Population = "
                          + speciesOfTheMonth.population);
    System.out.println("Growth rate = " +
                         speciesOfTheMonth.growthRate + "%");
}
```

这是用speciesOfTheMonth替换this后得到的。

关键字this就像一个等待着调用方法的对象来填充的空白一样。如果this是必需的，则必须非常频繁地使用它，因此Java允许省略this和它后面的点。但我们认为this和那个点是隐式存在的。这是一种常用的省略方式。程序员很少使用this参数，但确实在某些情况下需要使用它。

快速参考：this参数

给出方法定义时，可以用关键字this作为调用对象的名字。

自测题

1. 假设要在图4-4所示的程序中添加另一个名为speciesOfTheYear的物种对象，并假设你希望由用户输入数据，具体来说就是由用户来输入名字、种群规模和增长率。要实现这一目标需要向程序中添加什么代码？（提示：只需要3~4行代码。）
2. 假设Employee是一个类，包含了一个void方法readInput，而dilbert是Employee类的对象。因此，可以用下列形式来命名并创建dilbert：
```
Employee dilbert = new Employee();
```
写出用dilbert作为调用对象的readInput方法调用。方法readInput的圆括号中不需要包含信息。
3. 假设为了更方便地对物种进行分类，你想为世界上每个物种分配一个数字和一个名字。修改图4-3中SpeciesFirstTry类的定义，使其可以包含一个数字。数字是int类型的。（提示：主要是要添加一些内容。注意，你要做的部分工作就是向一些方法中添加内容。）
4. 假设你住在一个理想化的世界，那儿的每个物种中男性和女性成员的数量都完全一样。给出一个可以添加到图4-3所示的SpeciesFirstTry类定义中的方法定义，方法名为femalePopulation。方法femalePopulation会返回种群中女性的数量。如果种群规模为奇数，那么，进行配对之后还会剩下一个物种成员，我们假设那名成员是一名女性。例如，如果种群规模为6，就有3名男性和3名女性。如果种群规模为7，就有3名男性和4名女性。同样给出名为malePopulation的方法的定义，这个方法与femalePopulation类似，返回种群规模中男性的数量。（提示：方法的定义非常短。两个定义的主体有点儿不同。）
5. 用this参数重新编写图4-3中的方法writeOutput的定义。注意，定义的含义完全不会改变。只是用了一种略有不同的方式来编写它。（提示：你要做的就是在某些位置添加this和点。）
6. 用this参数重新编写图4-3中的方法readInput的定义。
7. 用this参数重新编写图4-3中的方法populationIn10的定义。
8. 图4-3中的方法PopulationIn10的定义中出现的(int)是什么意思，为什么要用到它？

4.1.7 局部变量

注意图4-3给出的方法populationIn10的定义。那个方法定义中包含了对变量populationAmount和count的声明。在方法内部声明的变量被称为**局部变量**（local variable）。因为这个变量的含义局限于这个方法定义，所以，称为局部的。如果有两个方法，每个方法都声明了一个同名的变量——比如，这两个变量都被称为populationAmount——它们会是两个碰巧同名的不同变量。对一个方法中名为populationAmount的变量的任何修改都不会对另一个方法中名为populationAmount的变量产生影响。就好像这两个方法是在不同的计算机上执行一样，或者就像计算机把其中一个方法中的populationAmount变量改成了populationAmount2一样。

由于程序的main部分自身也是一个方法，所以，main中声明的所有变量都是方法main的局部变量。如果其中一个变量碰巧和其他方法中声明的变量同名，那么，这两个变量就是碰巧同名的两个不同的变量。例如，图4-5所示的程序和类定义中，显示在图下半部分的程序的main方法中包含了对变量newAmount的声明，而上半部分的类定义中，类中的showNewBalance方法也声明了一个名为newAmount的变量。这是两个不同的变量，两个变量都叫作newAmount。main中名为newAmount的变量被设置为800.00。之后，有一个方法调用：

```
myAccount.showNewBalance();
```

如果去看看方法showNewBalance的定义，并做一点儿算术运算，你会发现在这个方法中，另一个名为newAmount的变量被设置为105.00。但这个变量对main中另一个名为newAmount的变量没有什么影响。进行了这次方法调用之后，main中名为newAmount的变量被写出，它的值仍为800.00。对方法showNewBalance中newAmount值的修改没有影响main中名为newAmount的变量。在这个例子中，这两个同名变量位于两个不同文件的不同定义中。但是，如果两个方法在同一个类定义，也就是在同一个文件中，情况也是一样的。

```
/**
 This class is used in the program LocalVariablesDemoProgram.
*/
public class BankAccount
{
    public double amount;
    public double rate;
    public void showNewBalance()
    {
        double newAmount = amount + (rate/100.0)*amount;
        System.out.println("With interest added the new amount is $"
                                         + newAmount);
    }
}
```

这个类定义保存在一个名为BankAccount.java的文件中。

两个名为newAmount的不同的变量。

这个程序存储在一个名为LocalVariableDemoProgram.java的文件中。

```
/**
 A toy program to illustrate how local variables behave.
*/
public class LocalVariablesDemoProgram
{
    public static void main(String[] args)
    {
        BankAccount myAccount = new BankAccount();
        myAccount.amount = 100.00;
        myAccount.rate = 5;
        double newAmount = 800.00;
        myAccount.showNewBalance();
        System.out.println("I wish my new amount were $" + newAmount);
    }
}
```

这并没有改变main中变量newAmount的值。

屏幕输出

```
With interest added the new amount is $105.0
I wish my new amount were $ 800.0
```

图4-5 局部变量

快速参考：局部变量

在一个方法定义内部声明的变量称为**局部变量**。如果两个方法中都有一个同名的局部变量，即使它们名字相同，也是不同的变量。

快速参考：全局变量

到目前为止，我们已经介绍过两种类型的变量了：含义局限于类对象的实例变量，以及含义局限于方法定义的局部变量。有些编程语言还有另一种类型的变量，称为**全局变量**（global variable），它的含义只局限于程序，即它根本没什么局限。Java中没有这种全局变量。

4.1.8 块

术语**块**（block）和**复合语句**（compound statement）实际上表示的是同样的内容，即一系列包含在花括号{}之间的Java语句。但这两个术语通常会用于不同的上下文中。在一个复合语句中声明一个变量时，通常会将复合语句称为块。

如果在一个块内（即在一个复合语句内）声明一个变量，这个变量就是局部于这个块的。这就意味着，块结束时，块中声明的所有变量就都消失了。在很多编程语言中，甚至可以用那个变量的名字去命名块外的其他变量。但是，在Java中，*一个方法定义中不能有两个同名的变量。*

在Java中，使用块内局部变量有时会有点儿麻烦。在Java中，不能将块内局部变量名重新用于其他的块外变量。因此，在块外声明变量有时会更简单一些。在块外声明的变量，即可以在块内使用，也可以在块外使用，而且不管是在块内还是块外，这些变量都具有相同的含义。

快速参考：块

块是复合语句，也就是花括号中包含的一系列语句的另一个名字。尽管块和复合语句是同样的，可是当花括号中包含变量声明时，还是倾向于使用术语块。块中声明的变量是局限于块的，因此当块的执行结束时，这些变量就消失了。但是，即使变量是局限于块的，也不能在同一个方法定义中将它们的名字用于其他内容。

▲ 易犯错误：块中声明的变量

在块中声明一个变量时，这个变量就是块的局部变量。这就意味着不能在块外使用这个变量。如果想在块外使用一个变量，就必须在块外声明。在块外声明的变量，既可以在块外使用，也可以在块内使用。△

■ Java提示：在`for`语句中声明变量

如下例所示，可以在for语句的初始化部分声明一个变量：

```
int sum = 0;
for (int n = 1; n <= 10; n++)
     sum = sum + n*n;
```

如果这样做，变量（在这个例子中就是n）就是**局限于`for`循环的**，不能在for循环之外使用。例如，下面的代码中，不能在System.out.println语句中使用n：

```
for (int n = 1; n <= 10; n++)
    sum = sum + n*n;
System.out.println(n);     //Invalid
```

这种特性有时是很讨厌而不是很有用的。而且，在不同的编程语言，甚至不同的Java版本中，对在for循环初始化部分声明的变量的处理是不同的。因此，我们宁愿不使用这个特性，而在for循环之外声明变量。但是，你可能会在其他程序员编写的代码中见到这种用法，所以应该了解这种特性。

4.1.9 基本类型的参数

图4-3定义的SpeciesFirstTry类中的方法populationIn10返回了一个物种在未来10年内的预期种群规模。但如果你希望得到未来5年或50年的预期值该怎么办呢？要是有一个方法，以表示年数的整数作为输入，并能返回未来相应年数的预期种群规模，就会很有用。要实现这项功能，需要通过某种方式在方法中留下一个空白，这样每次方法调用时都可以用不同的年数值来填充这个空白。方法中留作空白的地方称为形式参数，或简单地称为**形参**。它们比简单的空白稍复杂一些，但是，可以把它们当成方法调用时需要用一些值来填充的空白或占位符，是不会犯太大错误的。

图4-6中定义的SpeciesSecondTry类包含了一个名为projectedPopulation的方法，这个方法有一个名为years的形参。调用方法时，要给出你希望用来替代形参years的值。比如，如果将speciesOfTheMonth声明为SpeciesSecondTry类型的变量，就可以用如下形式通过方法projectedPopulation来计算12年内的种群规模了：

```
futurePopulation = speciesOfTheMonth.projectedPopulation(12);
```

```
import java.util.*;

public class SpeciesSecondTry
{
    public String name;
    public int population;
    public double growthRate;

    public void readInput()
    {
        <方法 readInput 的定义与图4-3中所示相同。>
    }
    public void writeOutput()
    {
        <方法 writeOutput 的定义与图4-3中所示相同。>
    }
    /**
     Returns the projected population of the calling object
     after the specified number of years.
    */
    public int projectedPopulation(int years)
    {
```

稍后在本章中会提到，应该用private来取代修饰符public。

图4-6 带有参数的方法

```
        double populationAmount = population;
        int count = years;
        while ((count > 0) && (populationAmount > 0))
        {
            populationAmount = (populationAmount +
                          (growthRate/100) * populationAmount);
            count--;
        }
        if (populationAmount > 0)
            return (int)populationAmount;
        else
            return 0;
    }
}
```

稍后在本章中将给出该类的一个更好的版本。

图4-6 （续）

在图4-7中，重新编写了图4-4中的程序，使用了SpeciesSecondTry类及其方法projectedPopulation。通过这个版本的类，可以预测未来任何年内的种群规模。甚至可以用一个变量来表示年数：

```
int projectedYears, futurePopulation;
System.out.println("Enter the projected number of years:");
projectedYears = keyboard.nextInt();
futurePopulation =
     speciesOfTheMonth.projectedPopulation(projectedYears);
System.out.println("In " + projectedYears + " years, the");
System.out.println("population will be " + futurePopulation);
```

```
/**
 Demonstrates the use of a parameter
 with the method projectedPopulation.
*/
public class SpeciesSecondTryDemo
{
    public static void main(String[] args)
    {
        SpeciesSecondTry speciesOfTheMonth = new SpeciesSecondTry();
        int futurePopulation;
        System.out.println("Enter data on the Species of the Month:");
        speciesOfTheMonth.readInput();
        speciesOfTheMonth.writeOutput();
        futurePopulation = speciesOfTheMonth.projectedPopulation(10);
        System.out.println("In ten years the population will be " +
                                                futurePopulation);
        speciesOfTheMonth.name = "Klingon ox";
        speciesOfTheMonth.population = 10;
        speciesOfTheMonth.growthRate = 15;
        System.out.println("The new Species of the Month:");
        speciesOfTheMonth.writeOutput();
        System.out.println("In ten years the population will be " +
                           speciesOfTheMonth.projectedPopulation(10));
    }
}
```

屏幕对话示例

对话与图4-4中完全相同。

图4-7 使用带有参数的方法

下面更仔细地看看方法projectedPopulation的定义。下面显示的方法头部中有些新的内容：

```
public int projectedPopulation(int years)
```

单词years被称为**形式参数**（formal parameter）或简单地称为**形参**（parameter）。形参用于方法定义，作为方法调用时插入值的替代。方法调用时插入的项被称为**实参**（argument）。在其他一些书中，也将实参称为**实际参数**（actual parameter）。比如，在下面的调用中，值10就是一个实参：

```
futurePopulation = speciesOfTheMonth.projectedPopulation(10);
```

使用上述方法调用时，*要在方法定义中所有出现形参的地方插入实参*（在这个例子中是10）来替代形参。在这个例子中，要在图4-6所示的方法projectedPopulation定义中插入实参10来替代形参years。之后，这个方法调用的执行就和前面所有的方法调用一样了。它会执行方法定义主体中的语句，直到遇到return语句为止。此时，就会将return语句的表达式中指定的值作为方法调用的返回值返回。

*要注意，在这个替换过程中只使用了实参的值，这是很重要的。如果方法调用中的实参是个变量，那么，插入的就是变量的值，而不是变量的名字。*比如，下列情况可能出现在某些使用了图4-6定义的SpeciesSecondTry类的程序中：

```
SpeciesSecondTry mySpecies = new SpeciesSecondTry();
int yearCount = 12;
int futurePopulation;
    futurePopulation = mySpecies.projectedPopulation(yearCount);
```

在这种情况下，插入图4-6中的方法projectedPopulation定义中替换形参years的是值12。替换years的不是变量yearCount。由于在这个过程中使用的只是实参的值，所以这种插入实参替换形参的方式称为**按值调用**（call-by-value）机制。在Java中，这是唯一一种可以用来替换基本类型形参的方式，比如int、double和char。正如最终会看到的，类类型的形参使用的替换机制有些不同，但现在关心的只是基本类型的形参和实参，如int、double和char。

这种形参替换方法的精确细节比现在讨论的要稍微复杂一些。通常，不需要关心这些精确的细节，但偶尔也需要了解这种替换机制到底是怎样工作的。因此，这里列出了精确的技术细节：*出现在方法定义中的形参是一个被初始化为实参值的局部变量。*实参是在方法调用的圆括号中给出的。比如，考虑下列方法调用：

```
futurePopulation =
   mySpecies.projectedPopulation(yearCount);
```

图4-6所示的方法projectedPopulation的形参years是方法projectedPopulation的局部变量，在这个方法调用中，将局部变量years设置为实参yearCount的值。效果等同于将方法定义的主体改成下列形式：

```
{
    years = yearCount;    这是插入实参yearCount后的效果。
    double populationAmount = population;
    int count = years;
    while ((count > 0) && (populationAmount > 0)
    {
        populationAmount = (populationAmount +
```

```
                    (growthRate/100) * populationAmount);
        count--;
    }
    if (populationAmount > 0)
        return (int)populationAmount;
    else
        return 0;
}
```

最后，要注意的是方法头部的形参是有类型的，例如，这里显示的形参years前面的类型是int：

```
public int projectedPopulation(int years)
```

每个形参都是有类型的，插入方法调用中用来替换形参的实参必须与形参的类型相匹配。因此，对方法projectedPopulation来说，方法调用的圆括号中给出的实参必须是int类型的。在实际应用中，这条规则并不像刚才所说的那么严格。在很多情况下，如果在方法调用中使用了一个与形参类型不匹配的实参，Java都会执行自动强制类型转换（类型映射）。例如，如果方法调用中实参的类型为int，而形参类型为double，Java就会将int类型的值转换成相应的double类型的值。下面的列表显示了会自动执行的强制类型转换。如果需要对方法调用实参进行转换以匹配于形参，下列任意一种类型的实参都会被自动转换成出现在其右侧的其他类型[①]：

```
byte --> short --> int --> long --> float --> double
```

注意，这和在第2章讨论过的、将一种类型的值存储到另一种类型的变量中去时采用的自动强制类型转换是完全一样的。因此，可以用一条更通用的原则来表述实参的自动强制类型转换和变量的自动强制类型转换：在Java需要列表中某种类型值的地方，都可以使用在列表中位于其左侧的任意类型的值。

到目前为止我们讨论的所有例子都是会返回值的方法，但我们讨论的所有关于形参和实参的原则都同样适用于void方法：void方法也可以有形参，它们的处理方式与会返回值的方法的处理方式完全相同。

快速参考：基本类型参数

形参是在方法定义起始处的方法名后面的圆括号中给出的。int、double或char这样的基本类型的形参都是局部变量。调用方法时，用方法调用中相应实参的值对形参进行初始化。这种机制称为**按值调用**参数机制。方法调用中的实参可以是2或'A'这样的字面常量，可以是变量，也可以是能够生成相应类型值的任意表达式。

注意，如果在方法调用中用一个基本类型的变量作为实参，那么，方法调用是无法改变这个变量的值的。

方法定义中可能（甚至经常）会有多个形参。在这种情况下，要在方法头部将每个形参都列出来，而且每个形参前面都要有一个类型。例如，一个方法定义的头部可能如下所示：

```
public void doStuff(int n1, int n2, double cost, char code)
```

① 如果形参是int类型或类型列表中int右边的其他数值类型，char类型的实参也会被转换，以匹配于数值类型。但不提倡使用这项特性。

注意，即使有多个同类型的形参，也必须在每个形参前加上一个类型名。

在方法调用中，圆括号中的实参数必须与方法定义头部的形参数完全相同。比如，对假想的方法doStuff的调用可能如下：

```
anObject.doStuff(42, 100, 9.99, 'Z');
```

如此例所示，这些形参是按序对应起来的。插入方法调用中的第一个实参是用来替换方法定义头部的第一个形参的，插入方法调用中的第二个实参是用来替换方法定义头部的第二个形参的，依此类推。除了前面介绍过的自动强制类型转换之外，每个实参的类型都必须与其对应的形参类型相匹配。

需要提请注意的一点是：类类型形参的行为与基本类型形参的行为有所不同。本章稍后将会介绍类类型形参。

快速参考：形参和实参之间的对应

形参是在方法定义起始处方法名后面的圆括号中给出的。在方法调用中，**实参**是在方法名后面的圆括号中给出的。方法调用中的实参的数量必须与相应方法定义中的形参的数量完全相同。

根据实参在圆括号内列表中所处的位置来确定它们是用来替换哪个形参的。插入方法调用中的第一个实参是用来替换方法定义中的第一个形参的，插入方法调用中的第二个实参是用来替换方法定义中的第二个形参的，依此类推。尽管在某些情况下当类型不匹配的时候，Java会执行自动强制类型转换，实参还是应该与其相应的形参具有相同的类型。

▲ 易犯错误：术语形参（parameter）和实参（argument）的用法

本书中的术语*形参*和*实参*的用法与其一般用法是一致的，但术语形参和实参通常是可以互换的。很多人会用术语参数（parameter）来表示所说的*形参*和*实参*。还有些人会用术语参数（argument）来表示所说的*形参*和*实参*。看到术语*参数*的时候，一定要根据上下文来确定它的确切含义。 △

4.1.10 类定义和方法定义语法小结

类定义的基本轮廓如下所示：

```
public class Class_Name
{
    Instance_Variable_Declaration_1
    Instance_Variable_Declaration_2
    ...
    Instance_Variable_Declaration_Last
    Method_Definition_1
    Method_Definition_2
    ...
    Method_Definition_Last
}
```

这是最常用的形式，但也可以把方法定义和实例变量声明交织在一起。

方法定义是由两个按下列顺序出现的部分组成的：

Method_Heading
Method_Body

到目前为止，所见过的方法头部都具有下列形式：

```
public Type_Name_Or_void Method_Name(Parameter_List)
```

*Parameter_List*由一系列形参名组成，每个形参名前面都有一个类型。如果列表中有多个项，项之间用逗号分隔。也可能根本就没有参数，那样的话圆括号中就是空的。

下面列出了一些方法头部示例：

```
public double Total(double price, double tax)
public void setValue(int count, char rating)
public void readInput()
public int projectedPopulation(int years)
```

*Method_Body*由一系列包含在花括号{}中的Java语句组成。如果方法要返回一个值，方法定义中就必须包含一条或多条return语句。

完整的类定义实例请参见图4-3和图4-6。

自测题

9. 未加限定的术语*parameter*和术语*formal parameter*有什么区别？
10. 定义一个可以添加到图4-6中的SpeciesSecondTry类定义的、名为density的方法。方法density有一个名为area的double类型参数。参数area用来表示物种所占的面积，以（平方英里）为单位。方法density会返回一个double类型的值，这个值等于每平方英里上此物种的个体数。可以假设面积总是大于零的。（提示：定义非常短。）
11. 定义一个可以添加到图4-6中的SpeciesSecondTry类定义的、名为fixPopulation的方法。这个方法有一个名为area的double类型参数。这个参数用来表示物种所占的面积，以平方英里为单位。方法fixPopulation修改了实例变量population的值，使每平方英里上都有一对个体存在。
12. 定义一个可以添加到图4-6中的SpeciesSecondTry类定义的、名为changePopulation的方法。这个方法有两个参数：一个参数是double类型的，名为area，用来表示物种所占的面积，以平方英里为单位；另一个参数是int类型的，名为numberPerMile，用来表示每平方英里上预期的个体数。方法changePopulation改变了实例变量population的值，使每平方英里上的个体数（近似）等于numberPerMile。

4.2　信息隐藏与封装

原因被隐藏起来，结果却是众所周知的。

——奥维德，古罗马作家，《变形记》

信息隐藏听起来好像是件坏事儿一样。把信息隐藏起来能有什么好处呢？计算机科学中使用的术语*信息隐藏*指的确实是一种真正意义上的信息隐藏，但在计算机科学中它被当作一种很好的编程技巧。其基本思想是，将某些类型的信息隐藏起来时，程序员的工作会变得更简单，而且程序员编写的代码也会更容易理解。本质上来说这是一种避免“信息超载”的手段。

4.2.1　信息隐藏

程序员不需要为了使用你定义的方法而去了解方法定义主体中的代码细节。如果一个方法（或其他软件）编写的很好，那么使用这个方法的程序员只需要知道方法完成了什么，而不需要操心方法是*如何*完成它的任务的。比如，你使用Scanner中的nextInt方法时，甚至都

不用看它的定义。并不是代码中包含了一些需要隐藏的秘密，关键在于查看那些代码对使用方法没什么帮助，只会使需要了解的内容变得更多，这样会分散编写程序的注意力。

设计一个方法，使用户不需要了解代码的精确细节就可以使用这个方法，这种方式称为**信息隐藏**，将其称为信息隐藏是为了强调方法的主体好像被藏起来，对程序员不可见一样。如果你觉得*信息隐藏*这个术语听起来太消极了，也可以使用*抽象*（abstraction）。在这情况下，*信息隐藏*和*抽象*这两个术语表示的是同一个意思。*抽象*一词应该不会让人感到惊奇。当对某样东西进行抽象时，就会丢失一些细节。例如，一篇论文或一本书的摘要就是对这篇论文或这本书的简要描述，而不是整本书或整篇论文的内容。

● 编程提示：参数名是局部于方法的

方法应该是自成一体的单元，它的设计应该独立于其他方法附带的细节，也应该独立于任何使用这个方法的程序。形参名就是这些附带细节中的一种。幸运的是，在Java中选择形参名的时候，不需要考虑形参名是否会与其他方法中的标识符相同。因为形参实际上是局部变量，它们的含义是局限于各自的方法定义的。

4.2.2 前置条件和后置条件注释

有些特定类型的注释称为前置条件和后置条件，通过这些注释来描述方法所完成的任务是一种高效且标准的方法。方法的**前置条件**（precondition）说明了一些在调用方法之前必须为真的条件。除非满足了前置条件，否则就不能使用方法，也不能期望方法能够正确执行。

后置条件（postcondition）描述了方法调用的效果。后置条件说明了在满足了前置条件的情况下，执行了方法之后，哪些内容会是真的。对一个会返回值的方法来说，后置条件会描述方法返回的值。对一个void方法来说，后置条件除了描述其他内容之外，还会描述对调用对象所作的所有修改。总的来说，后置条件描述了方法调用产生的所有效果。

例如，下面显示了一些适用于图4-3中的方法writeOutput的前置条件和后置条件：

```
/**
 Precondition: The instance variables of the calling
 object have values.
 Postcondition: The data stored in (the instance variables
 of) the calling object have been written to the screen.
 */
public void writeOutput()
```

图4-6中方法projectedPopulation的注释可以表示为下列形式：

```
/**
 Precondition: years is a nonnegative number.
 Postcondition: Returns the projected population of the
 calling object after the specified number of years.
 */
public int projectedPopulation(int years)
```

如果后置条件只对返回值进行了描述，程序员通常会将省略Postcondition。可以将前面的注释写成下列形式：

```
/**
 Precondition: years is a nonnegative number.
```

```
  Returns the projected population of the calling object
  after the specified number of years.
*/
public int projectedPopulation(int years)
```

有些设计规范会要求所有的方法都使用前置条件和后置条件。其他一些规范会在那些通过名字就可以明显反映出其行为的方法中省略显式的前置条件和后置条件。通常认为readInput、writeOutput和set这样的名字都是毋须解释的。但是，需要遵循的规则是：听从老师或导师的指导，不能确定的时候，就加上前置条件和后置条件。

有些程序员不愿意在注释中使用单词precondition和postcondition。但在写方法注释时，还是应该按照前置条件和后置条件来思考。实际上，最重要的并不是precondition和postcondition这两个单词，而是它们所表示的概念。

■Java提示：断言检测

断言（assertion）是用来描述程序状态的语句。断言可以为真也可以为假，如果程序不出错，就应该是真的。前置条件和后置条件注释就是断言的示例。在程序的其他地方也可以使用断言注释。比如，下列代码中所有的注释都是断言：

```
//n == 1
while (n < limit)
{
    n = 2*n;
}
//n >= limit
//n is the smallest power of 2 >= limit.
```

注意，虽然随着n和limit值的不同，这些断言都可以为真或为假，但如果程序正确执行，它们都应该为真。断言会“断定”（当程序执行到断言所处的位置时）程序代码中某些事情是真的。

在Java中，可以插入一个检测来查看一个断言是否为真，如果断言不为真，就停止运行程序并输出一条错误信息。Java中的**断言检测**具有下列形式：

assert *Boolean_Expression;*

如果用正确的方式编译并运行程序，执行断言检测时会发生下列情况：如果计算*Boolean_Expression*得*true*，什么事情都不会发生，如果计算*Boolean_Expression*得false，程序就会终止，并输出一条错误消息报告断言失败。

例如，可以将前面显示的代码写成下列形式，用断言检测取代其中的两个注释：

```
assert n == 1
while (n < limit)
{
    n = 2*n;
}
assert n >= limit
//n is the smallest power of 2 >= limit.
```

注意，我们只把3个注释中的两个翻译成了断言检测。不是所有的断言注释都能翻译成断言检测的。比如，最后一个注释就是一个断言，它可能为真也可能为假，如果程序代码正确，就为真。但是没有简单的方式可以将最后这条注释翻译成布尔表达式。要这样做并不是不可能的，但需要使用的代码可能比要检测的内容更复杂。是否要将最后这条注释翻译成断言检测，取决于特定情况的具体细节。

可以将断言检测打开，也可以将其关闭。调试代码时可以将其打开，这样失败的断言就会终止程序并输出一条错误消息。一旦代码调试好了，就可以将断言检测关闭，以使代码更有效地运行。

即使不打算在运行时将断言打开，也要用一种不同的方式来编译包含断言的类。所有代码编译完之后，运行程序时，也可以打开或关闭断言检测。

如果要用一条单行命令来编译类，可以按下列方式来编译一个带有断言检测的类：

```
javac -source 1.5 YourProgram.java
```

然后，运行程序时，可以打开或关闭断言检测。通常运行程序时都会关闭断言检测。要打开断言检测运行程序，可以使用下列命令：

```
java -enableassertions YourProgram
```

如果使用了IDE，就应该有一些设置断言检测选项的方式。查看所用IDE的文档。如果你用的是TextPad，为编译及运行代码打开断言检测的方法如下所示：在Configure菜单中选择Preferences，然后在Tools子菜单中选择Compile Java，并选中Prompt for Parameters复选框。在同一个Tools子菜单中，还可以找到Run Java Application命令，也需要为这条命令设置Prompt for Parameters选项①。然后，在编译类时，会出现一个窗口，可以在这个窗口中输入javac编译命令的实参比如-source1.5 "YourProgram.java"。类似地，运行程序时，也会出现一个窗口，可以在这个窗口中输入java运行命令的实参，比如-enableassertions YourProgram。窗口已经以正确格式显示了最后一个实参，如是否带引号，以及是否使用完整路径名等。只要加上-source 1.5或-enableassertions即可。

快速参考：断言检测

断言检测由关键字assert及跟随其后的布尔表达式和一个分号组成。可以在代码中的任何地方插入断言检测。如果断言检测打开了，且计算断言检测中的布尔表达式得到的是false，程序就会结束，并输出一条适当的错误消息。如果断言检测没有打开，断言检测就会被当作一条注释处理。

语法：

assert *Boolean_Expression*;

举例：

```
assert n >= limit;
```

4.2.3 public和private修饰符

将类的实例变量设置为public不是一种好的编程习惯。通常，所有的实例变量都使用修饰符private。在本节中，将介绍修饰符public和private之间的区别。

修饰符public表示任意其他的类或程序都可以直接访问并修改这个实例变量。例如，图4-7中的程序包含了下列3行代码，这些代码设置了对象speciesOfTheMonth中实例变量public的值：

```
speciesOfTheMonth.name = "Klingon ox";
speciesOfTheMonth.population = 10;
speciesOfTheMonth.growthRate = 15;
```

对象speciesOfTheMonth是SpeciesSecondTry类的一个实例，图4-6给出了它的定义。其中，实例变量name、population和growthRate的修饰符都是public，因此，前面这3条语句是完

① 如果运行的是applet，还需要为Tools子菜单中的Run Java Applet命令选中Prompt for parmeters选项。

全合法的。

现在，假设将图4-6中的SpeciesSecondTry类定义中实例变量name前面的修饰符public改为private，使类定义以下列形式开始：

```
public class SpeciesSecondTry
(
    private String name;
    public int population;
    public double growthRate;
```

进行这种修改之后，在图4-7的程序中使用下列语句就是非法的了：

```
speciesOfTheMonth.name = "Klingon ox";  //Invalid when private。
```

下面两条语句仍然是合法的，因为population和growthRate的修饰符仍然是public：

```
speciesOfTheMonth.population = 10;
speciesOfTheMonth.growthRate = 15;
```

如图4-8所示，将所有的实例变量都声明为private是一种很好的编程习惯。无论什么时候，只要在实例变量前加修饰符private，从类定义之外就不能访问这个实例变量的名字。在类定义中的任意一个方法内，都可以用任意方式使用这个实例变量名。尤其是，在类定义内可以直接修改实例变量的值。但在类定义之外，就无法直接引用这个实例变量名了。

```
import java.util.*;

public class SpeciesThirdTry
{
    private String name;
    private int population;
    private double growthRate;

    public void readInput()
    {
        Scanner keyboard = new Scanner(System.in);
        System.out.println("What is the species' name?");
        name = keyboard.nextLine();

        System.out.println("What is the population of the species?");
        population = keyboard.nextInt();
        while (population < 0)
        {
            System.out.println('Population cannot be negative.");
            System.out.println("Reenter population:");
            population = keyboard.nextInt();
        )

        System.out.println(
                    "Enter growth rate (percent increase per year):");
        growthRate = keyboard.nextDouble();
    }

    public void writeOutput()
     <方法 writeOutput 的定义与图4-3中所示相同。>
    /**
     Precondition: years is a nonnegative number.
     Returns the projected population of the calling object
     after the specified number of years.
    */
    public int projectedPopulation(int years)
    <方法 projectedPopulation 的定义与图4-6中所示相同。>
)
```

稍后在本章中将给出这个类的一个更好版本。

图4-8　带有private实例变量的类

例如，对于图4-8中显示的SpeciesThirdTry类，由于所有实例变量都被标记为private，所以，在任何程序中（或者除了SpeciesThirdTry类的方法之外的任何类方法定义中），下面的后三条语句都是非法的：

```
SpeciesThirdTry secretSpecies = new SpeciesThirdTry(); //Valid
secretSpecies.readInput(); //Valid
secretSpecies.name = "Aardvark"; //Invalid. name is private.
System.out.println(secretSpecies.population); //Invalid
              //population is private.
System.out.println(secretSpecies.growthRate); //Invalid
              // growthRate is private.
```

注意，方法readInput的调用是合法的。因此，尽管对象的实例变量是private的，仍然有办法可以对其进行设置。将实例变量设置为private并不意味着无法对其进行修改了。这只意味着无法用实例变量名直接引用变量了（除非是在包含这个实例变量的类定义内部）。

在同一个类的方法定义中，可以用任意方式访问私有实例变量。图4-8中所示的方法readInput的定义是用下列赋值语句来设置实例变量的值的：

```
name = keyboard.nextLine();
```

及

```
population = keyboard.nextInt();
```

在类的任何方法中，都可以以任意方式访问那个类所有的实例变量，即使是被标记为private的实例变量也一样。

类中的方法也可以是private的。如果方法被标记为private，就无法在类定义之外调用它，但仍然可以在同一个类的所有其他方法定义内调用它。大多数方法都被标记为public，但如果有一个方法仅用于类的其他方法定义内部，就可以将这个“辅助”方法标为private。

快速参考：public和private限定符

在一个类定义中，每个实例变量声明和每个方法定义前面都可以使用public或private。如果一个实例变量的前面是private，那么，除了同一个类的方法定义内部，在其他任何地方都无法用名字来引用这个变量；如果它前面是public，对实例变量名的使用就没有任何限制了。如果一个方法定义的前面是private，就不能在类定义之外调用这个方法；如果方法前面是public，对方法的使用就没什么限制了。

通常，会将所有实例变量都标识为private，而将大部分或所有方法都标识为public。

● 编程提示：实例变量应该标记为private

应该将类中所有实例变量都标识为private。因为这样可以强制使用这个类的程序员（无论是你自己还是其他人）只能通过方法来访问实例变量。这样，类就可以控制程序员访问实例变量的方式了。

将所有实例变量都标记为private确实控制了对实例变量的访问，但如果有合理的原因要访问实例变量该怎么办呢？在这些情况下，就应该提供一些访问方法。**访问方法**（accessor method）就是允许读取一个或多个实例变量中所含数据的方法。在图4-9中，又一次重新编写了物种的类。这个版本中包含了获取每个实例变量值的访问方法。这些访问方法都以get开头，如getName。

通过访问方法可以读取私有实例变量中的数据。还有其他一些被称为**设置方法**（mutator method）的方法，这些方法允许修改存储在私有实例变量中的数据。类定义中有一个名为set的设置方法，用来为实例变量设置新的值。图4-10中的程序说明了设置方法set的用法。这个程序与图4-7中的程序类似，但由于这个版本的物种类中含有私有实例变量，所以我们必须用设置方法set重新设置实例变量的值。

看起来，访问方法和设置方法好像使将实例变量设置为private的目的落空了，但这种做法是有一定道理的。设置方法可以对所作的修改是否恰当进行检测，如果有问题就向用户发出警告。例如，设置方法set会查看程序是否无意中将population设置为一个负数。

```
import java.util.*;

public class SpeciesFourthTry
{
     private String name;
     private int population;
     private double growthRate;
<这里放方法readInput、writeOutput和Projected Population的定义，它们与图4-3和图4-6中所示相同。>
    public void set(String newName,
                    int newPopulation, double newGrowthRate)
    {
        name = newName;
        if (newPopulation >= 0)
            population = newPopulation;
        else
        {
            System.out.println( "ERROR: using a negative population.");
            System.exit(0);
        }
        growthRate = newGrowthRate;
    }

    public String getName()
    {
        return name;
    }

    public int getPopulation()
    {
        return population;
    }

    public double getGrowthRate()
    {
        return growthRate;
    }
}
```

稍后会为这个类定义一个更好的版本。

设置方法会进行查看，以确保没有将实例变量设置为不恰当的值。

图4-9 带有访问方法和设置方法的类

```
import java.util.*;

/**
 Demonstrates the use of the mutator method set.
*/
public class SpeciesFourthTryDemo
{
    public static void main(String[] args)
    {
        SpeciesFourthTry speciesOfTheMonth = new SpeciesFourthTry();
        int numberOfYears, futurePopulation;

        System.out.println("Enter number of years to project:");
        Scanner keyboard = new Scanner(System.in);
        numberOfYears = keyboard.nextInt();

        System.out.println(
                       "Enter data on the Species of the Month:");
        speciesOfTheMonth.readInput();
        speciesOfTheMonth.writeOutput();
        futurePopulation =
             speciesOfTheMonth.projectedPopulation(numberOfYears);
        System.out.println("In " + numberOfYears
                                  + " years the population will be "
                                  + futurePopulation);
        speciesOfTheMonth.set("Klingon ox", 10, 15);
        System.out.println("The new Species of the Month:");
        speciesOfTheMonth.writeOutput();
        System.out.println("In " + numberOfYears
           + " years the population will be "
           + speciesOfTheMonth.projectedPopulation(numberOfYears));
    }
}
```

屏幕对话示例

```
Enter number of years to project:
10
Enter data on the Species of the Month:
What is the species' name?
Ferengie fur ball
What is the population of the species?
1000
Enter growth rate (percent increase per year):
-20.5
Name = Ferengie fur ball
Population = 1000
Gowth rate = -20.5%
In 10 years the population wioo be 100
The new Species of the Month:
Name = Klingon ox
Population = 10
Growth rate = 15.0%
In 10 years the population will be 40
```

图4-10 设置方法的使用

快速参考：访问和设置方法

从一个或多个私有实例变量中读取并返回数据的公有方法称为**访问方法**。访问方法名通常以get开头。

对存储在一个或多个私有实例变量中的数据进行修改的公有方法称为**设置方法**。设置方法名通常以set开头。

自测题

13. 在图4-10中，是以下列形式设置对象speciesOfTheMonth的数据的：

```
speciesOfTheMonth.set("Klingon ox", 10, 15);
```

可以用下列代码来替代吗？

```
speciesOfTheMonth.name = "Klingon ox";
speciesOfTheMonth.population = 10;
speciesOfTheMonth.growthRate = 15;
```

如果可以使用这些替换代码，为什么不用呢？如果不能使用这些替换代码，解释一下为什么不能使用。

14. 给出下列方法的前置条件和后置条件，这个方法是要添加到图4-9的SpeciesFourthTry类中去的：

```
public void updatePopulation()
{
    population = (int)(population
                + (growthRate/100)*population);
}
```

15. 什么是断言？给出一些断言的例子。

16. 假设Java中没有断言检测（早期的Java版本中就没有）。编写一些代码来模拟下列断言检测：

```
assert balance > 0
```

balance是一个double类型的变量。

17. 什么是访问方法？什么是设置方法？

18. 给出一个名为Person的类的完整定义，这个类有两个实例变量，一个表示人的名字，另一个表示人的年龄。按照图4-9中的模式，在类中包含访问方法和设置方法，还要包含输入和输出方法。类中再没有其他方法。

编程示例：Purchase类

图4-11包含了一个用于单次购物的类，比如买12个苹果或2夸脱（1夸脱=1.136升）牛奶。该类是一个超市收银台程序的一部分。通常超市给出的价格都不是单位价格，即超市给出的不是一件物品的价格，而是几件物品的价格，比如，5个1.25美元，或者3个1.00美元。他们希望把苹果标成5个1.25美元，你就会买5个而不是2个苹果。但5个1.25美元实际上就是每个0.25美元，如果你买2个苹果，他们只收0.50美元。

所需的实例变量如下所示：

```
private String name;
private int groupCount; //Part of price,
                        //like the 2 in 2 for $1.99.
private double groupPrice; //Part of price,
                        //like the $1.99 in 2 for $1.99.
private int numberBought; //Total number being purchased.
```

```
import java.util.*;

/**
 Class for the purchase of one kind of item, such as 3 oranges.
 Prices are set supermarket style, such as 5 for $1.25.
*/
public class Purchase
{
    private String name;
    private int groupCount; //Part of price, like the 2 in 2 for $1.99.
    private double groupPrice;
                       //Part of price, like the $1.99 in 2 for $1.99.
    private int numberBought; //Total number being purchased.

    public void setName(String newName)
    {
        name = newName;
    }

    /**
     Sets price to count pieces for $costForCount.
     For example, 2 for $1.99.
    */
    public void setPrice(int count, double costForCount)
    {
        if ((count <= 0) || (costForCount <= 0))
        {
            System.out.println("Error: Bad parameter in setPrice.");
            System.exit(0);
        }
        else
        {
            groupCount = count;
            groupPrice = costForCount;
        }
    }

    public void setNumberBought(int number)
    {
        if (number <= 0)
        {
            System.out.println("Error: Bad parameter in setNumberBought.");
            System.exit(0);
        }
        else
            numberBought = number;
    }

    /**
     Gets price and number being purchased from keyboard.
    */
    public void readInput()
    {
        Scanner keyboard = new Scanner(System.in);
        System.out.println("Enter name of item you are purchasing:");
        name = keyboard.nextLine();
```

图4-11 Purchase类

```
        System.out.println("Enter price of item as two numbers.");
        System.out.println("For example, 3 for $2.99 is entered as");
        System.out.println("3 2.99");
        System.out.println("Enter price of item as two numbers, now:");
        groupCount = keyboard.nextInt();
        groupPrice = keyboard.nextDouble();
        while ((groupCount <= 0) || (groupPrice <= 0))
        {//Try again:
            System.out.println(
                    "Both numbers must be positive. Try again.");
            System.out.println("Enter price of item as two numbers.");
            System.out.println("For example, 3 for $2.99 is entered as");
            System.out.println("3 2.99");
            System.out.println("Enter price of item as two numbers, now:");
            groupCount = keyboard.nextInt();
            groupPrice = keyboard.nextDouble();
        }

        System.out.println("Enter number of items purchased:");
        numberBought = keyboard.nextInt();
        while (numberBought <= 0)
        {//Try again:
            System.out.println("Number must be positive. Try again.");
            System.out.println("Enter number of items purchased:");
            numberBought = keyboard.nextInt();
        }
    }

    /**
     Outputs price and number being purchased to screen.
    */
    public void writeOutput()
    {
        System.out.println(numberBought + " " + name);
        System.out.println("at " + groupCount+ " for $" + groupPrice);
    }

    public String getName()
    {
        return name;
    }

    public double getTotalCost()
    {
        return ((groupPrice/groupCount)*numberBought);
    }

    public double getUnitCost()

    {
            return (groupPrice/groupCount);
```

图4-11 （续）

```
    }

    public int getNumberBought()
    {
        return numberBought;
    }
}
```

图4-11 （续）

用一个例子来解释这些实例变量的含义是最简单的。如果以5个苹果1.25美元的价格买12个苹果，name的值就是"apples"，groupCount的值为5，groupPrice的值为1.25，numberBought的值则为12。注意，5个苹果1.25美元的价格是存储在两个实例变量groupCount（存储5）和groupPrice（存储1.25美元）中的。

比如，对于方法getTotalCost。购物的总花销计算如下：

```
(groupPrice/groupCount) * numberBought
```

或者具体来说，如果以5个苹果1.25美元的价格买12个苹果，总花销为：

```
(1.25/5) * 12
```

同样，注意一下方法readInput、setPrice和setNumberBought。所有这些方法都在不应该出现负数的地方对负数进行了检测，比如用户输入购买数量的时候。图4-12给出了一个使用了这个类的简单演示程序。

```
public class PurchaseDemo
{
    public static void main(String[] args)
    {
        Purchase oneSale = new Purchase();
        oneSale.readInput();
        oneSale.writeOutput();
        System.out.println("Cost each $" + oneSale.getUnitCost());
        System.out.println("Total cost $"
                                + oneSale.getTotalCost());
    }
}
```

屏幕对话示例

```
Enter name of item you are purchasing:
pink grapefruit
Enter price of item as two numbers.
For example, 3 for $ 2.99 is entered as
3 2.99
Enter price of item as two numbers, now:
4 5.00
Enter number of items purchased:
0
Number must be positive . Try again.
Enter number of items purchased:
2
2 pink grapefruit
at 4 for $5.0
Cost each $1.25
Total cost $2.5
```

图4-12 Purchase类的使用

4.2.4 封装

在第1章中曾说过，**封装**（encapsulation）是一种将类定义中对理解类对象的使用没有必要帮助的细节全部隐藏起来的过程。要使封装发挥效用，就要以这样一种方式给出类定义，即程序员无需关心类定义的内部细节即可使用它。在关于信息隐藏的介绍中，已经讨论过一些可以实现这一目标的技术了。封装是信息隐藏的一种形式。正确的封装可以清楚地将一个类定义分成两部分，称为用户接口[①]和实现。**用户接口**（user interface）会告诉程序员使用类时需要了解的全部内容。用户接口由公有方法的头部、类的已定义常量、用来告诉程序员如何使用类中的公有方法和公有已定义常量的注释组成。在程序中使用这个类时需要了解的全部内容都应该包含在类定义的用户接口部分中。

实现（implementation）包含了这个类定义中所有的私有元素，主要是类的私有实例变量，以及公有和私有方法的定义。注意，在Java代码中，类定义的用户接口和实现并没有被分隔开，而是混杂在一起的。比如，图4-11就突出显示了`Purchase`类的用户接口。尽管在运行一个使用了某个类的程序时，需要用到那个类的实现部分，但在编写使用那个类的代码时，并不需要了解任何与实现有关的内容。

在定义使用了封装原则的类时，必须确实在概念上将用户接口和实现完全分开，这样接口才能成为一种简单且安全的类的描述方式。对这个问题来说，一种方式就是想像在实现和接口之间有一堵墙，它们可以穿过这堵墙进行良好的通信。图4-13以图形方式显示了它们之间的关系。通过封装可以将实现和用户接口清晰地分开，用这种方式定义类时，可以说类进行了**良好封装**（well encapsulated）。

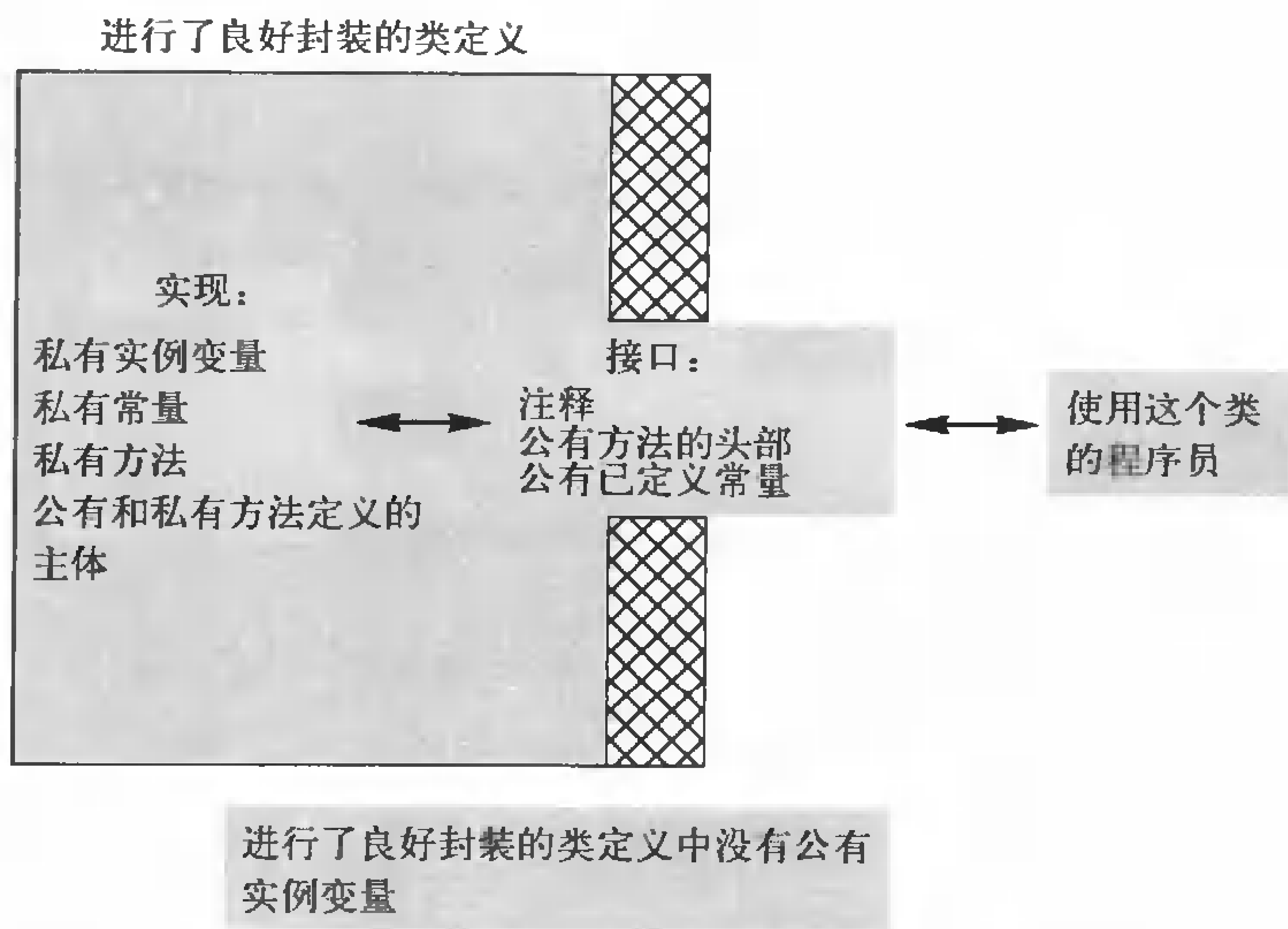


图4-13　封装

下面列出了定义一个封装良好的类时需要遵循的一些最重要的原则：

(1) 在类定义之前放置一条注释，用来告诉程序员应该如何认识这个类的数据和方法。

① 接口这个词在Java语言中还有技术含义。当我们说用户接口时，这个单词的用法会略有不同，尽管在本质上，接口这个词的两种用法是一样的。

(注意，这条注释不一定是实例变量的列表。如果类中描述了一个钱款数，程序员就应该把它当作美元和美分，而不应该在用double类型的实例变量来记录钱款数时，把它当成一个double类型的实例变量；也不应该在用两个int类型的实例变量来记录美元和美分的数额时，把它们当成两个int类型的实例变量。实际上，使用类的程序员就不应该关心钱款是用double类型的实例变量来表示的，还是用两个int类型的实例变量或者其他什么方式来表示的。)

(2) 类中所有的实例变量都应该标记为private。

(3) 提供公有的访问方法和设置方法来读入或修改对象中的数据。同样，还要为程序员提供一些操作类中的数据时所需的其他公有方法，比如，应该提供一些输入和输出方法。

(4) 在方法头部之前放置一条注释，对每个公有方法的使用进行完整的说明。

(5) 将每个辅助方法都标记为private。

(6) 类定义中有些注释对如何使用这个类进行了描述，这些注释是用户接口的一部分。这些注释放在类定义之前时，通常用来描述一些通用特性；放在特定的方法定义之前时，通常用来描述如何使用特定方法。只有在对实现进行解释的时候，才需要添加其他注释。编写注释时需要遵循的一个原则是：为用户接口注释使用/**/类型的注释，为实现中的注释使用//类型的注释。图4-11突出显示了用户接口注释。

用封装的形式定义类时，应该能够在不对使用类的程序进行任何修改的情况下，回过头来对类定义中的实现细节进行修改。这是一种很好的测试方法，可以用来测试你是否编写了一个封装良好的类定义。对类定义的实现细节进行修改的原因通常会有很多。比如，可能找到了一种更有效的方式来实现一个方法，使这个方法的调用运行得更快。你甚至可能会在不改变方法调用方式及其基本功能的基础上，对实现的某些细节进行修改。比如，如果有一个用于银行账户对象的类，就可能会修改账户透支时收取的罚金数。

常见问题：什么是API?

术语**API**表示的是*应用编程接口*（application programming interface）。类的**API**和类的用户接口本质上是一样的。阅读与类库有关的文档时，经常会看到API一词。

常见问题：什么是ADT?

术语**ADT**是*抽象数据类型*（abstract data type）的缩写。ADT是一种用良好的信息隐藏技术编写的数据类型。因此在Java中，ADT和封装良好的类定义基本上是一个意思。

快速参考：封装

封装是描述现代编程技术时经常会用到的一个术语。**封装**意味着将数据和动作组合到一个单一项（在这里，就是一个类对象）中，并将实现的细节隐藏起来。因此，*信息隐藏*、ADT和*封装*的总体思想基本上是一致的。用操作性很强的术语来说，它们的基本思想就是：使用类的程序员无需了解类的实现细节。

4.2.5　用javadoc自动生成文档

如果Java安装程序来自Sun公司（甚至其他一些地方），其中就会包含一个名为javadoc的程序，这个程序可以自动为类的用户接口生成文档。这个文档会告诉那些使用程序或类的人，在使用过程中需要了解哪些内容。要想获得更有用的javadoc文档，就必须以一种特定的方式来编写注释。对本书中所有类作的注释都可以用于javadoc（尽管由于空间的限制，注释要比理想情况下稍微稀疏一些）。如果按照图4-11中的注释方式，对类定义进行了正确的注释，javadoc就会将类定义作为输入，并为类的用户接口生成一份具有良好格式的展示文档。比如，如果对图4-11中的类定义运行javadoc，输出中只会包含那些突出显示的文本（还会对空格和换行符等进行调整。）

要理解本书的内容，并不需要使用javadoc。也不需要为了编写Java程序使用javadoc。而且，要阅读javadoc生成的文档，一定要使用Web浏览器（或其他HTML查看程序）。但是，如果你已经在使用Web浏览器了，比如Netscape Navigator或Internet Explorer，很可能会发现javadoc既很好用又很有用。附录I对javadoc进行了介绍。

4.2.6　UML类图

在本章开头给出了一个类图的例子（图4-2）。现在你学习的知识已经足够理解其中所有的表示法了。但我们不去看那个类图，而是来看一个新的类图。图4-14包含了图4-11中Purchase类的类图。除了加减号之外，类图中的细节都已经非常清楚了。实例变量或方法前面的加号（+）表示这个变量或方法是公有的。实例变量或方法前面的减号（-）表示它们是私有的。

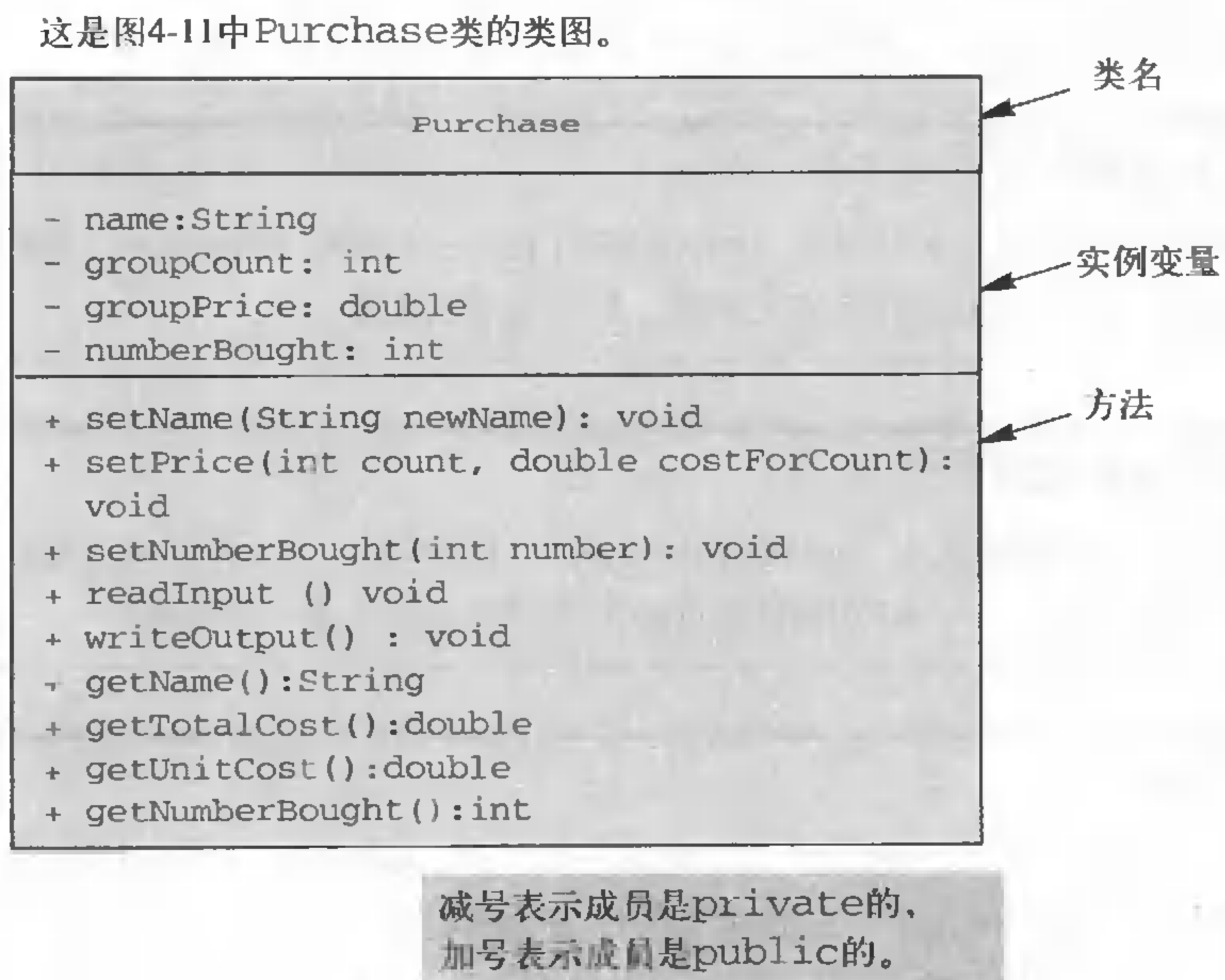


图4-14　UML类图

注意，类图包含的内容比类接口要多，但比完整的实现要少。通常，类图都是在定义一

个类之前完成的。它是接口和实现的轮廓图。类图主要是给定义类的程序员使用的。而接口则是给编写其他软件时会用到这个类的程序员使用的。

自测题

19. 什么是封装良好的类定义？
20. 什么时候应该将类定义中的实例变量标记为private，什么时候应该将其标记为public？
21. 在什么情况下会将一个方法标记为private？
22. 在类定义中，标记为private的内容会是用户接口的一部分吗？
23. 在类定义中，方法定义的主体会是用户接口的一部分吗？

4.3 对象与引用

"你好像很伤心，"骑士不安地说，"让我唱支歌来安慰安慰你吧。"

"很长吗？"爱丽丝问，因为这一天里她已经听了很多诗歌了。

"它虽然长，"骑士说，"但是非常非常精彩。听了我唱的歌，有的人流泪，有人就……"

"就怎么样？"爱丽丝问，因为骑士突然不说了。

"有的人就不流泪。歌的名字叫《鳕鱼的眼睛》。"

"哦，那是歌的名字吗？"爱丽丝问道，想做出很感兴趣的样子。

"不，你不明白，"骑士有点急躁地说，"那是别人叫的名字，它真正的名字是《上年纪的人》。"

"那么我应该说《别人叫的名字》吗？"爱丽丝纠正自己说。

"不，不应该；这完全是另一回事儿！这支歌还被称作《方法和手段》。不过你知道，这也是别人叫的。"

"噢，那这歌到底叫什么呢？"爱丽丝说，现在她完全糊涂了。

"我正要说呢。这歌真正的名字叫做《在门上歇一下》；调子是我创作的。"骑士说。

——刘易斯·卡洛尔，《爱丽丝镜中奇遇记》

类类型的变量，如图4-12中的oneSale，与int、double和char这样的基本类型变量的表现有很大的不同。类类型的变量是其所属类的对象名，但是，对象并不是变量的值，在这一点上，它与"数字6是int类型变量的值"是不同的。可以用类类型的变量命名一个对象，但命名的过程会有些微妙之处。本节将介绍如何用类类型的变量来命名对象，以及与类类型的方法参数在Java中的表现有关的内容。

4.3.1 类类型的变量与对象

类类型的变量命名对象的方式，与int或char这样的基本类型变量存储值的方式不同。不管是基本类型还是类类型的变量，所有的变量都是作为存储单元实现的。如果变量是基本类型的，变量的值就存储在分配给那个变量的存储单元中。但是，如果变量是类类型的，变量命名的对象会被存储在内存的其他某个单元中，命名对象的变量中存储的则是对象所处的内存地址。

这就是基本类型的变量和类类型的变量会以不同的方式来命名值的原因。对一个基本类型，比如int类型的值来说，存储一个值时需要的内存空间通常都是相同的。在Java中，int类型是有最大值的，因此int类型值的长度是有限的。但是，类类型的对象，比如String类

的对象，可以是任意长的。String类型变量的存储单元是定长的，无法存储任意长的字符串。但它可以存储任意一个字符串的地址的，因为总会有最后一个地址，所以，用来表示一个地址的字节数总是有限的。

存储对象的内存地址称为对象的**引用**（reference）。

类类型的变量中包含的是引用，这会引发一些令人惊奇的结果。类类型的变量与基本类型变量的表现有很大的不同。下列代码可以作为一个程序main部分的开始：

```
SpeciesFourthTry klingonSpecies, earthSpecies;
klingonSpecies = new SpeciesFourthTry();
earthSpecies = new SpeciesFourthTry();
int n, m;
n = 42;
m = n;
```

正如你所预期的，有两个int类型的变量：n和m。两者的值都为42，但如果对其中一个进行了修改，另一个的值仍然会是42。例如，如果接下来程序为

```
n = 99;
System.out.println(n + " and " + m);
```

程序产生的输出将为

```
99 and 42
```

到目前为止还没什么令人吃惊的情况发生，现在假设接下来的程序为：

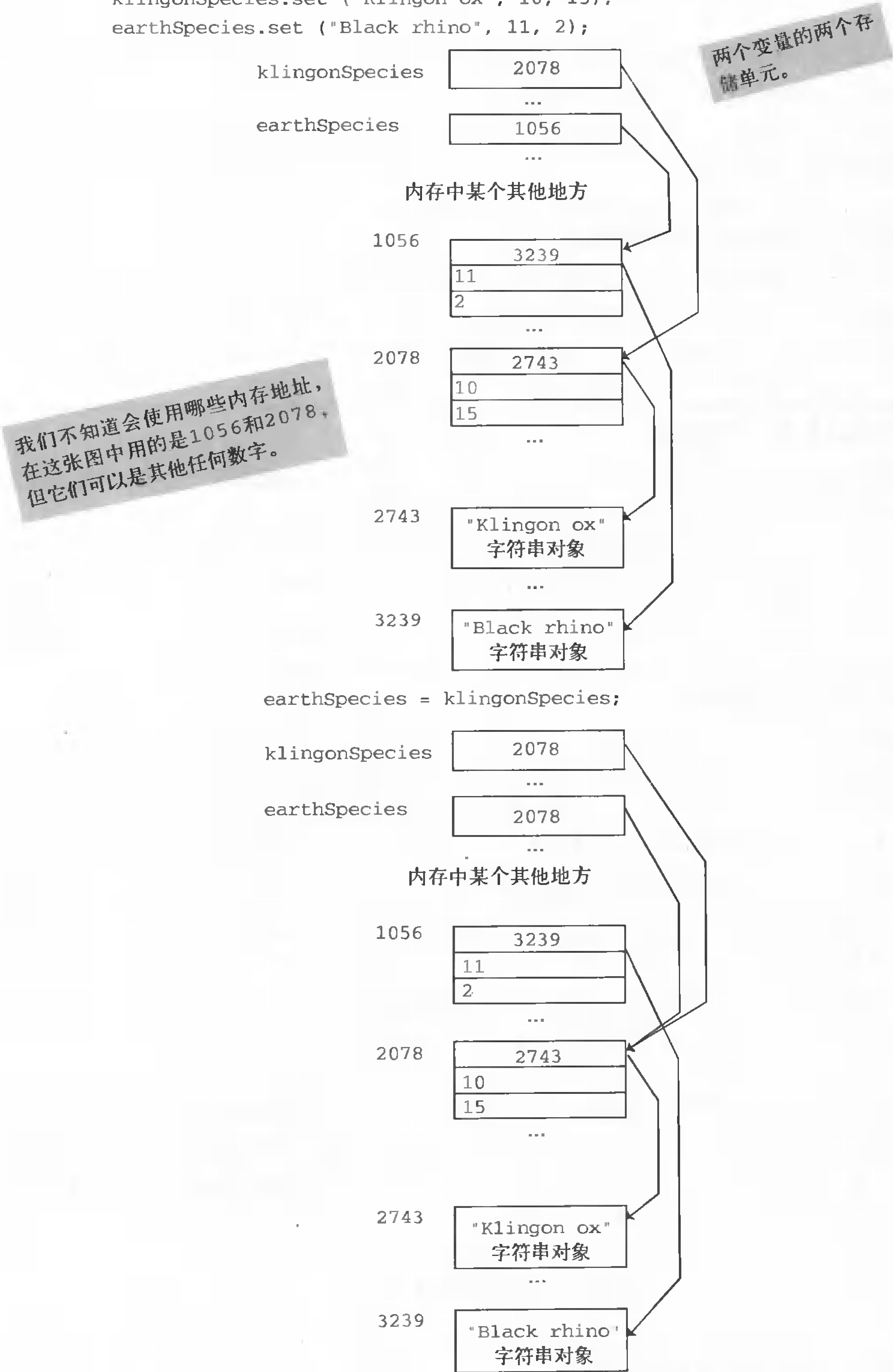
```
klingonSpecies.set("Klingon ox", 10, 15);
earthSpecies.set("Black rhino", 11, 2);
earthSpecies = klingonSpecies;
earthSpecies.set("Elephant", 100, 12);
System.out.println("earthSpecies:");
earthSpecies.writeOutput();
System.out.println("klingonSpecies:");
klingonSpecies.writeOutput();
```

你可能认为klingonSpecies会是Klingon ox，而earthSpecies会是elephant，但产生的输出可能会让你很吃惊。输出如下所示：

```
earthSpecies:
Name = Elephant
Population = 100
Growth rate = 12%
klingonSpecies:
Name = Elephant
Population = 100
Growth rate = 12%
```

发生了什么？有两个变量klingonSpecies和earthSpecies，但只有一个对象。这两个变量中包含了相同的引用，因此，这两个变量命名的是同一个对象。因为它们是同一个对象，所以，修改klingonSpecies时，也就修改了earthSpecies；修改earthSpecies时，也就修改了klingonSpecies。

每个对象都存储在计算机内存的某个单元中，并且那个单元会有一个地址。变量earthSpecies和klingonSpecies确实只是一些普通的变量（像int变量一样），但它们存储了SpeciesFourthTry类的对象的内存地址。*说一个类类型的变量命名了一个对象时，就是指变量中包含了那个对象的内存地址*，如图4-15所示。



klingonSpecies.set ("Klingon ox", 10, 15);
earthSpecies.set ("Black rhino", 11, 2);
两个变量的两个存储单元。
klingonSpecies
2078
earthSpecies
1056
内存中某个其他地方
1056
3239
11
2
2078
2743
10
15
我们不知道会使用哪些内存地址，在这张图中用的是1056和2078，但它们可以是其他任何数字。
2743
"Klingon ox"
字符串对象
3239
"Black rhino"
字符串对象
earthSpecies = klingonSpecies;
klingonSpecies
2078
earthSpecies
2078
内存中某个其他地方
1056
3239
11
2
2078
2743
10
15
2743
"Klingon ox"
字符串对象
3239
"Black rhino"
字符串对象

图4-15 类变量

赋值语句

```
earthSpecies = klingonSpecies;
```

只是将klingonSpecies中的内存地址复制到变量earthSpecies中，这样，它们就具有了同样的内存地址，因此也就命名了同一个对象。

关于内存地址的提醒：内存地址是一个数，但它不是int值那样的数。因此不要试图把它当成一个普通的整数来处理。

记住：类类型的变量存储的是内存地址

基本类型的变量存储的是类型的值。类类型变量的表现与之不同。类类型的变量存储的并不是类的对象，而是对象在计算机内存中所处的内存地址。这样，类类型的变量就可以作为类的对象名来使用了。但是，有些运算符，比如=和==，对类类型变量和基本类型变量的表现是完全不同的。

记住：内存地址是数，也不是数

类类型的变量中存储的是内存地址。内存地址是一个数字。但不能像使用一个存储了数字的变量那样使用类类型的变量。内存地址的一个重要特性是它标识了存储单元。实现这项功能的人使用了数字，而不是字母、颜色或任何其他东西来标识存储单元，这只是一种偶然的特性。Java要防止你利用这种偶然特性，以防止你去做一些不应该做的事情，比如获取对受限内存的访问，否则会把计算机搞得一团糟。这样也会使代码更易懂。

快速参考：类类型和引用类型

类类型的变量表示的实际上并不是类的对象。类类型的变量中只是内存中存储对象的地址。这种内存地址通常称为对内存中对象的**引用**。因此，通常将类类型称为引用类型。**引用类型**只是一种在其变量中包含了引用（即包含了内存地址），而不是对象实际值的类型。但是，除了类类型之外，还有其他的引用类型，因此在提到类名时，会使用类类型（class type）一词。所有的类类型都是引用类型，但还存在其他一些不是类类型的引用类型，参见第6章。

常见问题：什么是new？

类类型变量的工作方式与基本类型变量不同。基本类型的变量中含有的是类型的值。类类型的变量实际包含的并不是那个类的对象，而是那个对象在内存中的地址。声明

```
SpeciesFourthTry s;
```

创建了一个可以包含一个内存地址的变量s。此时，程序中有了一个可以存储内存地址的空间，但还没有空间可以用来存储SpeciesFourthTry类型对象的实例变量中的数据。要想获得用来存储实例变量值的存储单元，程序就要使用new。下列语句为一个SpeciesFourthTry类型的对象分配了一个存储单元，并将那个存储单元的地址放在了变量s中：

```
s = new SpeciesFourthTry();
```

从一种非正式的角度来看，可以认为new创建了对象的实例变量。

▲ 易犯错误：对类类型变量使用 = 和= =

在前面的小节中，对类类型的变量使用赋值运算符时，会得到一些令人吃惊的结果。其相等性测

试的表现看起来好像也比较奇怪。假设SpeciesFourthTry类是如图4-9所示定义，并假设程序中包含下列代码：

```
SpeciesFourthTry klingonSpecies = new SpeciesFourthTry();
SpeciesFourthTry earthSpecies = new SpeciesFourthTry();
klingonSpecies.set("Klingon ox", 10, 15);
earthSpecies.set("Klingon ox", 10, 15);
if (klingonSpecies == earthSpecies)
System.out.println("They are EQUAL.");
else
System.out.println("They are NOT equal.");
```

这些代码会产生下列输出：

```
They are NOT equal.
```

图4-16说明了这段代码的执行过程。 △

图4-16 为对象使用==带来的危险

问题在于，尽管直觉上认为这两个物种是相等的，但一个类类型的变量实际上只包含了一个内存地址。而内存中有两个SpeciesFourthTry类型的对象。它们表示的是现实世界中的同一个物种，但却有着不同的内存地址，而==运算符只查看内存地址是否相等。==运算符对一种类型的相等性进行了测试，但这并不是你通常关心的相等性。定义类时，通常应该为类定义一个名为equals的方法，用来测试对象是否相等。

■ Java提示：为类定义一个equals方法

用==运算符对两个对象进行比较时，查看的是它们是否具有相同的内存地址。测试的并不是直觉上认为的"相等"。要对直觉意义上的相等性进行测试，就应该定义一个名为equals的方法。在图4-17中，最后一次重新编写了物种类的定义，这次，我们添加了一个名为equals的方法。这个equals方法是用于Species类对象的，其用法与用于String类型对象的String类方法equals完全一样。图4-18中的程序演示了对equals方法的使用。

在Species类的equals 方法定义中使用了String类中的方法equalsIgnoreCase。正如在第2章指出的那样，这个方法是作为Java语言的一部分自动提供的。如果除了有些字母在一个字符串中是大写，在另一个字符串中是小写之外，被比较的两个字符串完全相同，equalsIgnoreCase就返回true；否则，就返回false。

注意，图4-17中的equals方法返回的总是true或false，因此返回值的类型为boolean。这条return语句看起来可能有点儿奇怪，但其实它只是一个可能会在if-else语句中用到的那种布尔表达式。如果注意到图4-17中的equals定义可以用下列伪代码来表示，会有助于理解这段伪代码：

```
如果 (( this.name.equalsIgnoreCase(otherObject.name))
                 && (this.population == otherObject.population)
                 && (this.growthRate == otherObject.growthRate))
那么返回 true
否则返回 false
```

因此，下列（来自图4-18中的程序）代码：

```
if (s1.equals(s2))
    System.out.println("Match with the method equals.");
else
    System.out.println("Do Not match with the method equals.");
```

等效于下列伪代码：

```
if 为true
    (s1.name.equalsIgnoreCase(s2.name))
    && (s1.population == s2.population)
    && (s1.growthRate == s2.growthRate), then
    System.out.println("Match with the method equals.");
else
    System.out.println("Do Not match with the method equals.");
```

在图4-17中的equals定义中，不需要使用参数this。其中给出的定义与下列代码是等效的：

```
public boolean equals(Species otherObject)
{
    return ((name.equalsIgnoreCase(otherObject.name))
```

```
            && (population == otherObject.population)
            && (growthRate == otherObject.growthRate))
}
```

```
import java.util.*;

public class Species
{
    private String name;
    private int population;
    private double growthRate;

    <readInput、writeOutput和projectedPopulation的方法定义放在这里，它们的定义与图4-3
        和图4-6中的定义是一样的。>
    <set、getName、getPopulation和getGrowthRate的方法定义放在这里，它们的定义与图4-9中
        的定义是一样的。>

    public boolean equals(Species otherObject)
    {
        return ((name.equalsIgnoreCase(otherObject.name))
                && (this population == otherObject.population)
                && (this growthRate == otherObject.growthRate));
    }
}
```

equalsIgnoreCase是String类的一个方法，是作为Java语言的一部分自动提供的。

图4-17 定义一个equals方法

实例变量population自身的含义和this.population的含义一样的。类似地，对其他任何实例变量自身的处理都和它前面有this和点的处理是一样的。

```
public class SpeciesEqualsDemo
{
    public static void main(String[] args)
    {
        Species s1 = new Species(), s2 = new Species();
        s1.set("Klingon Ox", 10, 15);
        s2.set("Klingon Ox", 10, 15);
        if (s1 == s2)
            System.out.println("Match with ==.");
        else
            System.out.println("Do Not match with ==.");
        if (s1.equals(s2))
            System.out.println("Match with the method equals.");
        else
            System.out.println(
                            "Do Not match with the method equals.");
        System.out.println(
                    "Now we change one Klingon Ox to all lowercase.");
```

图4-18 equals方法的演示

```
            s2.set("klingon ox", 10, 15);
            if (s1.equals(s2))
                System.out.println("Still match with the method equals.");
            else
                System.out.println(
                                "Do Not match with the method equals.");
    }
}
```

屏幕输出

```
Do Not match with ==.
Match with the method equals.
Now we change one Klingon Ox to all lowercase.
Still match with the method equals.
```

图4-18 （续）

4.3.2节将对返回boolean类型值的方法进行更多的介绍。

没有一种永远可用的独特的equals定义。equals定义的编写取决于你打算如何使用这个类。图4-17中的定义表明，如果Species类的两个对象表示的是相同的记录，即具有相同的物种名、相同的种群规模以及相同的增长率，就认为这两个对象是相等的。在其他一些情况下，你可能希望用equals将两个具有相同物种名，但种群规模或增长率可能不同的对象定义为相等的。这可能对应于这样一种情况：如果两个对象是同一物种的记录，即使是同一物种在不同时期的记录，也认为它们是相等的。

应该一直用标识符equals作为你创建的、用来测试两个对象是否相等的方法名。不要使用其他标识符，比如same；甚至也不要使用equal（没有s）。这是由于Java中的某些软件是根据确切的名字equals来测试对象的相等性的。这个软件会调用名为equals的方法，所以，最好将编写的方法命名为equals。

如果没有为类定义equals方法，Java会自动创建一个默认的equals定义，但这个定义的表现很可能和你的预期有所不同。所以，最好还是定义一个自己的equals方法。

编程示例：物种类

图4-19给出了物种对象类的最后一个版本。它和图4-17中的定义是一样的，但这次的版本包含了所有细节，这样就可以看到一个完整的示例了。还编写了不使用参数this的equals方法定义，因为大多数程序员都会使用这种形式。图4-19中的equals定义完全等效于图4-17中的定义。图4-20包含了Species类的类图。

```
import java.util.*;

/**
 Class for data on endangered species.
*/
```

这个类定义与图4-17是一样的，只是这里显示了所有的细节。

图4-19 完整的Species类

```
public class Species
{
    private String name;
    private int population;
    private double growthRate;

    public void readInput()
    {
        Scanner keyboard = new Scanner(System.in);
        System.out.println("What is the species' name?");
        name = keyboard.nextLine();

        System.out.println("What is the population of the species?");
        population = keyboard.nextInt();
        while (population < 0)
        {
            System.out.println("Population cannot be negative.");
            System.out.println("Reenter population:");
            population = keyboard.nextInt();
        }

        System.out.println("Enter growth rate (percent increase per year):");
        growthRate = keyboard.nextDouble();
    }

    public void writeOutput()
    {
         System.out.println("Name = " + name);
         System.out.println("Population = " + population);
         System.out.println("Growth rate = " + growthRate + "%");
    }

    /**
     Precondition: years is a nonnegative number.
     Returns the projected population of the calling object
     after the specified number of years.
    */
    public int projectedPopulation(int years)
    {
        double populationAmount = population;
        int count = years;
        while ((count > 0) && (populationAmount > 0))
        {
            populationAmount = (populationAmount +
```

图4-19 （续）

```
                        (growthRate/100) * populationAmount);
            count--;
        }
        if (populationAmount > 0)
            return (int)populationAmount;
        else
            return 0;
    }

    public void set(String newName, int newPopulation,double newGrowthRate)
    {
        name = newName;
        if (newPopulation >= 0)
            population = newPopulation;
        else
        {
            System.out.println("ERROR: using a negative population.");
            System.exit(0);
        }
        growthRate = newGrowthRate;
    }

    public String getName()
    {
        return name;
    }

    public int getPopulation()
    {
        return population;
    }

    public double getGrowthRate()
    {
        return growthRate;
    }

    public boolean equals(Species otherObject)
    {
        return ((name.equalsIgnoreCase(otherObject.name))
                && (population == otherObject.population)
                && (growthRate == otherObject.growthRate));
    }
}
```

这个equals版本等效于图4-17中的版本。在图4-17中，显式地使用了this参数。这个版本省略了this参数，但我们认为this参数是隐式存在的。

图4-19 （续）

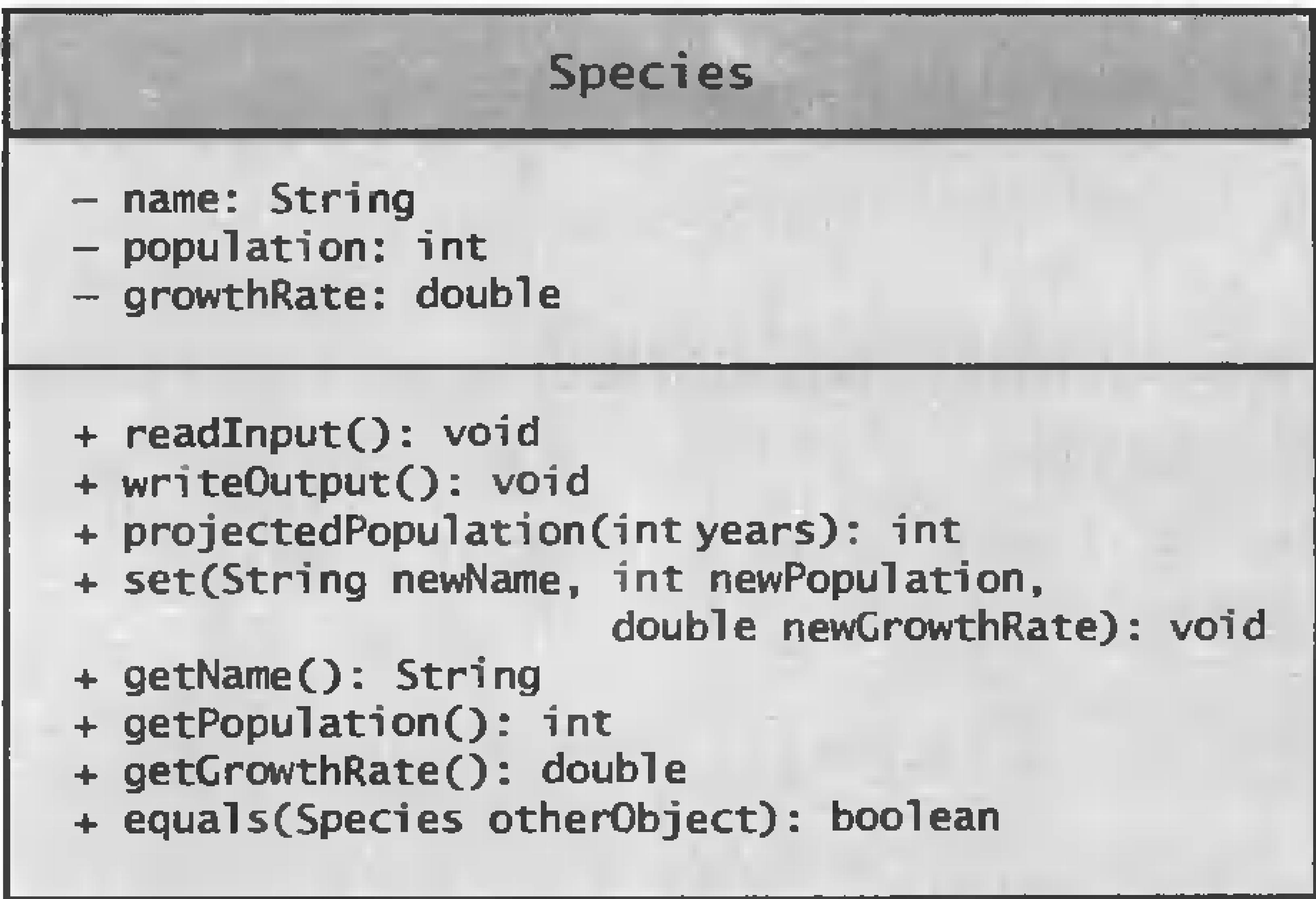


图4-20 图4-19中的Species类的类图

4.3.2 返回布尔值的方法

方法可以返回一个boolean类型的值。这并不是新鲜事：只要指定一个boolean返回类型，并在return语句中使用一个布尔表达式就可以了，如图4-19中Species类的equals方法，下面再现了这个方法：

```
public boolean equals(Species otherObject)
{
    return ((name.equalsIgnoreCase(otherObject.name))
        && (population == otherObject.population)
        && (growthRate == otherObject.growthRate))
}
```

这个方法只是计算了return语句的布尔表达式。那个布尔表达式会生成一个为true或false的值。而方法equals会将这个值返回。

可以在一条if-else语句、while语句或其他需要布尔表达式的语句中调用equals方法。也可以将方法equals或任何其他返回布尔值的方法返回的值，存储到一个boolean类型的变量中。例如：

```
Species s1 = new Species(), s2 = new Species();
<一些用来设置s1和s2值的代码>
boolean areEqual;
areEqual = s1.equals(s2);
<其他代码>
if (areEqual)
    System.out.println("They are equal.");
else
    System.out.println("They are not equal.");
```

可以将下列方法添加到图4-19所示的Species类定义，作为另一个返回布尔值的方法示例：

```
/**
 Precondition: The calling object and the argument
 otherSpecies both have values for their population.
 Returns true if the population of the calling object
 is greater than the population of otherSpecies;
 otherwise, returns false.
```

```
*/
public boolean largerPopulationThan(Species otherSpecies)
{
    return (population > otherSpecies.population);
}
```

然后，就可以用与equals方法相同的方式来使用largerPopulationThan方法了。例如，在某些程序中可能会出现下列代码：

```
Species s1 = new Species(), s2 = new Species();
<一些用来设置s1和s2值的代码>
if (s1.largerPopulationThan(s2))
    System.out.println(s1.getName() + " has the larger population.");
else
    System.out.println(s2.getName() + " has the larger population.");
```

作为一个附加示例，还可以将下列方法添加到图4-19所示的Species类定义中：

```
/**
 Precondition: The calling object has a value for
 its population.
 Returns true if the population of the calling object
 is zero; otherwise, returns false.
*/
public boolean isExtinct()
{
    return (population == 0);
}
```

然后，在某个程序中可能会出现下列示例代码：

```
Species s1 = new Species();
<一些用来设置s1值的代码>
if (s1.isExtinct())
    System.out.println(s1.getName() + " is extinct.");
else
    System.out.println(s1.getName() + " is still with us.");
```

自测题

24. 什么是引用类型？类类型是引用类型吗？基本类型（如int）是引用类型吗？
25. 对两个类类型的对象进行比较，看它们是否“相等”时，应该使用==还是equals方法呢？
26. 对两个int类型的对象进行比较，看它们是否“相等”时，应该使用==还是equals方法呢？
27. 为一个可以添加到图4-19所示的Species类中的、名为largerGrowthRateThan的方法编写一个方法定义。方法largerGrowthRateThan有一个Species类型的实参。如果调用对象的增长率高于实参给出的增长率，方法就返回true；否则，就返回false。

4.3.3 类类型参数

对类类型参数的处理和对基本类型参数的处理是不同的。从某种意义上说，在讨论对类类型对象使用赋值运算符时，已经讨论过这种区别了。下面两点内容有助于描述类类型参数是如何工作的：

（1）首先，回想一下，对类来说，赋值运算符是怎样工作的：对类类型的对象使用赋值

运算符时，复制的实际上是内存地址。假设Species就是图4-19中定义的那个类，考虑下列代码：

```
Species species1 = new Species();
Species species2 = new Species();
species2.readInput();
species1 = species2;
```

正如在前面的讨论过的，现在species1和species2就是同一个对象的两个名字。

(2) 考虑一下基本类型的参数是如何工作的。例如，图4-10中使用的对方法projected-Population的调用：

```
futurePopulation =
    speciesOfTheMonth.projectedPopulation(numberOfYears);
```

projectedPopulation的方法定义在图4-19中给出。定义以如下形式开始：

```
public int projectedPopulation(int years).
{
    double populationAmount = population;
    int count = years;
    while ((count > 0) && (populationAmount > 0))
    {
          ⋮
```

回想一下，形参years实际上就是一个局部变量。调用方法projectedPopulation时，这个局部变量years被初始化为实参numberOfYears的值。因此，调用方法时，就好像将下列赋值语句暂时插入到方法定义中了：

```
years = numberOfYears;
```

换句话说，就好像在这个方法调用的过程中，方法projectedPopulation的定义被改成了下列形式：

```
public int projectedPopulation(int years)
{
    years = numberOfYears;
    double populationAmount = population;
    int count = years;
    while ((count > 0) && (populationAmount > 0))
    {
          ⋮
```

开头很长，但如果理解了这两点，类类型参数的工作就很容易解释了。类类型参数的工作方式和第2点中所述的基本类型参数的工作方式相同，但是因为对类类型变量来说，赋值运算符具有不同的含义，所以效果也有很大的不同！①

用稍有不同的措辞（但含义相同）再来解释一遍。下面是在图4-18中使用的对方法equals的调用：

```
if (s1.equals(s2))
    System.out.println("Match with the method equals.");
else
```

① 有些程序员把类类型参数的参数机制称为引址调用（call-by-reference）参数传递。另外一些人则认为使用这个术语是不正确的。问题在于引址调用的常用定义不止一个。但有一点是明确的：Java中的类类型参数与其他语言中被称为引址调用参数的参数行为有所不同。因此，在这里不使用引址调用一词。无论怎么称呼它，在任何情况下，理解类类型参数的工作机制才是最重要的。

```
    System.out.println("Do Not match with the method equals.");
```

在这个调用中，s2是图4-19中定义的一个Species类类型的实参。在这里再现了方法equals的定义（这是图4-17给出的equals版本，等效于图4-19中的版本）：

```
public boolean equals(Species otherObject)
{
    return (( this.name.equalsIgnoreCase(otherObject.name))
        && (this.population == otherObject.population)
        && (this.growthRate == otherObject.growthRate));
}
```

在s1.equals(s2)中调用equals方法时，就好像在方法定义的起始处临时插入了下列赋值语句：

```
otherObject = s2;
```

换句话说，在这个equals调用过程中，方法定义等效于下列代码：

```
public boolean equals(Species otherObject)
{
    OtherObject = s2;//You cannot do this, but
            //Java acts as if you could and did do this.
    return (( this.name.equalsIgnoreCase(otherObject.name))
            && (this.population == otherObject.population)
            && (this.growthRate == otherObject.growthRate));
}
```

但是，回想一下，这条赋值语句只是将s2的内存地址复制到变量otherObject中去了，otherObject只是成了s2命名的对象的另一个名字。因此，对名为otherObject的对象所做的任何事情实际上都是对名为s2的对象做的。因此，上述代码的执行效果就像方法执行了下列动作一样：

```
return (( this.name.equalsIgnoreCase(s2.name))
        && (this.population == s2.population)
        && (this.growthRate == s2.growthRate));
```

注意，对类类型的参数而言，对形参（这个例子中的otherObject）执行的任何动作实际上都是对方法调用中的实参（这个例子中的s2）执行的。因此，方法调用实际作用并修改的是方法调用中使用的实参。

对方法equals来说，这种参数传递机制对类类型参数产生的效果和对基本类型参数产生的效果没什么太大的区别。但对其他一些方法来说，两者的区别就非常明显了。4.3.4节将用一个明显的例子来说明类类型参数与基本类型参数的区别。

快速引用：类类型的参数

形参是在方法定义起始处方法名后面的圆括号中给出的。类类型的形参就是一个含有了类类型对象内存地址的局部变量。调用方法时，这个形参会被初始化为方法调用中相应实参的地址。用技术性不太强的术语来说，这就意味着形参会被当作方法调用中相应实参给出的对象的替换名来使用。

注意，这就意味着，如果在方法调用中使用的是类类型的实参，方法调用就可以改变那个实参。

4.3.4　类类型参数和基本类型参数的比较

假设如图4-21所示，向Species类中添加了一个名为makeEqual的方法，形成了一个新的

名为DemoSpecies的类。这个类只是用于演示的，因此不需要考虑类定义的其余部分。注意，方法makeEqual有个DemoSpecies类型的参数，而且makeEqual方法修改了这个参数。

```
import java.util.*;

/**
 This is a version of the class Species, but is only a toy
 example designed to demonstrate the difference between
 parameters of a class type and parameters of a primitive type.
*/
public class DemoSpecies
{
    private String name;
    private int population;
    private double growthRate;

    /**
     Precondition: Calling object has been given values.
     Postcondition: otherObject has the same data as the
     calling object. The calling object is unchanged.
    */
    public void makeEqual(DemoSpecies otherObject)
    {
        otherObject.name = this.name;
        otherObject.population = this.population;
        otherObject.growthRate = this.growthRate;
    }

    /**
     Tries to set intVariable equal to the population of
     the calling object. But it cannot succeed, because
     arguments of a primitive type cannot be changed.
    */
    public void tryToMakeEqual(int intVariable)
    {
        intVariable = this.population;
    }

    public boolean equals(DemoSpecies otherObject)
    <equals方法定义的其余部分与图4-19中所示相同。>
    <类定义的其余部分与图4-19中Species类的定义相同。>
}
```

图4-21 只是一个演示类

图4-22中所示的演示程序中。对makeEqual的调用中有一个DemoSpecies类型的实参s2。注意，方法主体中的修改实际上都是对实参s2执行的。方法的确可以改变一个类类型实参的值。

现在来看一下同样出现在图4-21中的名为tryToMakeEqual的方法。注意，这个方法有一个基本类型int型的参数，而且这个tryToMakeEqual方法修改了形参。图4-22中的演示程序用int类型的实参aPopulation调用了tryToMakeEqual。注意，方法主体中执行的修改对实

参aPopulation没有什么影响。这是因为Java对基本类型的实参使用的是按值调用参数机制，同时也是因为基本类型的变量中包含的是实际值，而不是内存地址。所以，形参就是一个包含了实参值的局部变量，所有的修改都会作用于这个局部变量，而不是实参。

```
public class ParametersDemo
{
    public static void main(String[] args)
    {
        DemoSpecies s1 = new DemoSpecies(),
                    s2 = new DemoSpecies();
        s1.set("Klingon Ox", 10, 15);
        s2.set("Ferengie Fur Ball", 90, 56);
        System.out.println("Value of s2 before call to method:");
        s2.writeOutput();
        s1.makeEqual(s2);
        System.out.println("Value of s2 after call to method:");
        s2.writeOutput();
        int aPopulation = 42;
        System.out.println("Value of aPopulation before call to method: "
                     + aPopulation);
        s1.tryToMakeEqual(aPopulation);
        System.out.println("Value of aPopulation after call to method: "
                     + aPopulation);
    }
}
```

屏幕输出

```
Value of s2 before call to method:
Name = Ferengie Fur Ball
Population = 90
Growth Rate = 56.0%
Value of s2 after call to method:
Name = Klingon Ox
Population = 10
Growth Rate = 15.0%
Value of aPopulation before call to method:42
Value of aPopulation after call to method:42
```

类类型的实参是可以改变的。

基本类型的实参是无法改变的。

图4-22 对类类型和基本类型的参数进行比较

类类型参数的用途比基本类型参数的用途要广泛。基本类型的参数可以用来向方法传递值，但方法不能修改任何作为实参传递给它的基本类型变量的值。另一方面，类类型的参数不仅可以用来向方法传递信息，方法还可以对类类型实参命名的对象进行修改。

记住：基本类型和类类型参数之间的区别

方法无法改变作为方法实参使用的基本类型变量的值，但方法可以改变一个类类型实参的实例变量的值。

自测题

28. 以下列代码作为起始的程序会有什么问题（Species类是在图4-19中定义的）？

```
public class SpeciesEqualsDemo
{
    public static void main(String[] args)
    {
        Species s1, s2;
        s1.set("Klingon Ox", 10, 15);
        s2.set("Klingon Ox", 10, 15);
        if (s1 == s2)
            System.out.println("Match with ==.");
        else
            System.out.println("Do Not match with ==.");
    }
}
```

29. 基本类型参数和类类型参数之间最大的区别是什么？

30. 下列程序会产生什么样的输出（Species类是在图4-19中定义的）？

```
public class ExerciseProgram
{
    public static void main(String[] args)
    {
        Species s1 = new Species();
        ExerciseClass mysteryMaker = new ExerciseClass();
        int n = 0;
        s1.set("Hobbit", 100, 2);
        mysteryMaker.mystery(s1, n);
        s1.writeOutput();
        System.out.println("n = " + n);
    }
}
```

ExerciseClass类如下所示：

```
public class ExerciseClass
{
    public void mystery(Species s, int m)
    {
        s.set("Klingon Ox", 10, 15);
        m = 42;
    }
}
```

31. 重新定义自测题18中的类Person，使其包含一个equals方法。

4.4 图形编程补充（选读）

整体大于部分之和。

——谚语

前面已经对方法和参数进行了完整的解释，现在回过头来看看前面介绍的图形编程补充材料，对一些内容进行更完整的解释。要通过方法以一种更清晰的方式来重写前面的一个图形applet，要对Graphics类进行更完整的解释，还要介绍其他一些绘图方法。最后，将介绍方法init，这是另一个与paint类似、但有着不同目的的applet方法。

4.4.1 Graphics类

Graphics类的对象表示的是屏幕上的一个区域，但它所表示的内容又不止如此。Graphics类的对象中包含了一些方法，这些方法可以在它所表示的屏幕区域中画图和书写文本。图4-23给出了其中一些方法的摘要。前面已经介绍过使用其中大部分方法的applet实例了。但在这段简要的介绍中至少存在两个明显的问题：Graphics对象是如何表示屏幕上的一个区域的？在一个applet的paint方法中应该插入什么样的Graphics对象作为参数？（在我们的applet示例中，这个参数通常称为canvas。）

drawOval

在点(*X*, *Y*)上画出一个具有指定长轴（宽度）和短轴（高度）的椭圆的轮廓。

语法：

Graphics_Object.drawOval(*X*, *Y*, *Width*, *Height*);

举例：

```
canvas.drawOval(155, 95, 10, 20);
```

fillOval

与drawOval一样，只是填充了椭圆。

语法：

Graphics_Object.fillOval(*X*, *Y*, *Width*, *Height*);

举例：

```
canvas.fillOval(155, 95, 10, 20);
```

drawArc

画出一条弧线，即画出椭圆的一部分。

语法：

Graphics_Object.drawArc(*X*, *Y*, *Width*, *Height*, *Start_Angle*, *Degrees_Shown*);

细节参见1.4节。

举例：

```
canvas.drawArc(150, 175, 100, 50, 180, 180);
```

fillArc

与drawArc一样，但椭圆的可见部分是实心的。

语法：

Graphics_Object.fillArc(*X*, *Y*, *Width*, *Height*, *Start_Angle*, *Degrees_Shown*);

举例：

```
canvas.fillArc(150, 175, 100, 50, 180, 180);
```

drawRect

在点（*X*, *Y*）上画出一个具有指定宽度和高度的矩形的轮廓。

语法：

Graphics_Object.drawRect(*X*, *Y*, *Width*, *Height*);

举例：

```
canvas.drawRect[i3](155, 95, 10, 20);
```

图4-23　Graphics类中的方法

fillRect

与drawRect一样，只是矩形被填充满了。

语法：

Graphics_Object.fillRect(*X*, *Y*, *Width*, *Height*);

举例：

```
canvas.fillRect (155, 95, 10, 20);
```

drawLine

在点(*X*1, *Y*1)和(*X*2, *Y*2)之间画一条线。

语法：

Graphics_Object.drawLine(*X*1, *Y*1, *X*2, *Y*2);

举例：

```
canvas.drawLine(0, 0, 50, 75);
```

drawString

从点(*X*, *Y*)开始写下指定的字符串。

语法：

Graphics_Object.drawString(*String*, *X*, *Y*);

举例：

```
canvas.drawString("World's Greatest applet.", 50, 50);
```

setColor

为后继的绘画（及书写）设置颜色。颜色一直有效，直到再次调用setColor将其改变为止。

语法：

Graphics_Object.setColor(*Color_Object*);

举例：

```
canvas.setColor(Color.RED);
```

图4-23 （续）

Graphics对象中有一些实例变量，这些实例变量包含了对Graphics对象所表示的屏幕区域的详细说明。在我们的例子中，Graphics对象通常表示对应于applet内部范围的区域。我们不需要关心这种表示方法的具体细节，我们需要了解的全部内容就是对一个applet来说，相关的Graphics对象以某种方式表示了applet的内部区域即可。

问题仍然存在：applet的这个Graphics对象从何而来呢？答案是，当运行一个applet时，就会自动创建一个适当的Graphics对象，（自动）调用paint方法时，就会将这个对象作为applet的paint方法的实参使用。这些都是自动发生的。这就是代码复用的益处。你可能需要很频繁地生成这个Graphics对象，并调用paint方法，所以applet库代码就帮你把所有这些工作都做了。通过这种方式，创建Graphics对象和调用paint方法的代码只需要写一遍，而不需要在每次为applet编写代码时都写一遍。怎样才能把这些库代码添加到你的applet定义中呢？这可以通过扩展JApplet（extends JApplet）来实现，不过更详细的解释要留到第7章介绍。

快速参考：Graphics类

Graphics类的对象表示了屏幕上的一块区域，还包含了一些可以在相应屏幕区域中画图及书写文字的方法。

applet的paint方法有一个Graphics类型的参数。运行applet时，会自动生成一个表示applet内部区域的Graphics对象，并会自动调用以这个Graphics对象为实参的paint方法。

自测题

32. 假设用下列重新编写的代码取代了图3-18中的paint方法。（标识符canvas全部被g替换了。）这样会对图3-18中的applet产生什么影响？

```
public void paint(Graphics g)
{
  //Draw face circle:
  g.setColor(Color.YELLOW);
  g.fillOval(X_FACE, Y_FACE, FACE_DIAMETER, FACE_DIAMETER);
  g.setColor(Color.BLACK);
  g.drawOval(X_FACE, Y_FACE, FACE_DIAMETER, FACE_DIAMETER);
  //Draw eyes:
  g.setColor(Color.BLUE);
  g.fillOval(X_RIGHT_EYE, Y_RIGHT_EYE, EYE_WIDTH, EYE_HEIGHT);
  g.fillOval(X_LEFT_EYE, Y_LEFT_EYE, EYE_WIDTH, EYE_HEIGHT);
  //Draw nose:
  g.setColor(Color.BLACK);
  g.fillOval(X_NOSE, Y_NOSE, NOSE_DIAMETER, NOSE_DIAMETER);
  //Draw mouth:
  g.setColor(Color.RED);
  g.drawArc(X_MOUTH, Y_MOUTH, MOUTH_WIDTH, MOUTH_HEIGHT,
                    MOUTH_START_ANGLE, MOUTH_DEGREES_SHOWN);
}
```

33. 在带有paint方法定义的applet中，比如图3-18和图3-20，没有对方法paint的调用。那paint方法怎么会对applet中的绘画有所影响呢？毕竟，如果方法没有调用，方法中的动作是不会被执行的。

编程示例：通过一个辅助方法重新绘制的多张脸

在图4-24中重新实现了图3-20的applet。在这个新版本中，将除了嘴和皮肤颜色之外的所有脸部绘画代码都分离出来，并将这些代码放在一个名为drawFaceSansMouth的方法的主体中。在这个applet较老的版本（图3-20）中，方法主体的代码被重复了3次（一次是在for循环的主体中，两次是在for循环之后）。在图4-24中，这些代码只在drawFaceSansMouth方法的主体中给出了一次。这节省了一些代码输入时间，但更重要的是，使得代码更易读了。方法drawFaceSansMouth将除了嘴和皮肤颜色之外所有复杂的脸部绘画任务都封装到它里面去了。理解

```
drawFaceSansMouth(canvas, i);
```

要比理解

```
g.setColor(Color.BLACK);
g.drawOval(X_FACE0 + 50*i, Y_FACE0 + 30*i,
                    FACE_DIAMETER, FACE_DIAMETER);
//Draw eyes:
g.setColor(Color.BLUE);
```

```
g.fillOval(X_RIGHT_EYE0 + 50*i, Y_RIGHT_EYE0 + 30*i, EYE_WIDTH, EYE_HEIGHT);
g.fillOval(X_LEFT_EYE0 + 50*i, Y_LEFT_EYE0 + 30*i, EYE_WIDTH, EYE_HEIGHT);
//Draw nose:
g.setColor(Color.BLACK);
g.fillOval(X_NOSE0 + 50*i, Y_NOSE0 + 30*i, NOSE_DIAMETER, NOSE_DIAMETER);
```

更容易。

当然，在设计方法时，前面那段较长的代码必须理解一次，但在老版本（图3-20）中，要理解3次。

注意，由于drawFaceSansMouth方法只是paint方法的辅助方法，因此，我们将drawFaceSansMouth方法设置成了私有方法。

```
import javax.swing.*;
import java.awt.*;

public class MultipleFaces extends JApplet
{
    public static final int FACE_DIAMETER = 50;
    public static final int X_FACE0 - 10;
    public static final int Y_FACE0 = 5;
    public static final int EYE_WIDTH = 5;
    public static final int EYE_HEIGHT = 10;
    public static final int X_RIGHT_EYE0 = 20;
    public static final int Y_RIGHT_EYE0 = 15;
    public static final int X_LEFT_EYE0 = 45;
    public static final int Y_LEFT_EYE0 = Y_RIGHT_EYE0;

    public static final int NOSE_DIAMETER = 5;
    public static final int X_NOSE0 = 32;
    public static final int Y_NOSE0 = 25;

    public static final int MOUTH_WIDTH = 30;
    public static final int MOUTH_HEIGHT0 = 0;
    public static final int X_MOUTH0 = 20;
    public static final int Y_MOUTH0 = 35;
    public static final int MOUTH_START_ANGLE = 180;
    public static final int MOUTH_DEGREES_SHOWN = 180;

    /**
     g is the drawing area. i is a parameter for the position of the face.
     As i increases the face is drawn lower and further to the right.
    */
    private void drawFaceSansMouth(Graphics g, int i)
    {
        g.setColor(Color.BLACK);
```

图4-24 为重复发生的子任务使用一个方法

```
    g.drawOval(X_FACE0 + 50*i, Y_FACE0 + 30*i,
                          FACE_DIAMETER, FACE_DIAMETER);
    //Draw eyes:
    g.setColor(Color.BLUE);
    g.fillOval(X_RIGHT_EYE0 + 50*i, Y_RIGHT_EYE0 + 30*i,
                              EYE_WIDTH, EYE_HEIGHT);
    g.fillOval(X_LEFT_EYE0 + 50*i, Y_LEFT_EYE0 + 30*i,
                              EYE_WIDTH, EYE_HEIGHT);
    //Draw nose:
    g.setColor(Color.BLACK);
    g.fillOval(X_NOSE0 + 50*i, Y_NOSE0 + 30*i,
                              NOSE_DIAMETER, NOSE_DIAMETER);
}

public void paint(Graphics canvas)
{
    int i;
    for (i = 0; i < 5; i++)
    {//Draw one face:
        if (i%2 == 0)//if i is even
        {//Make face yellow
            canvas.setColor(Color.YELLOW);
            canvas.fillOval(X_FACE0 + 50*i, Y_FACE0 + 30*i,
                            FACE_DIAMETER, FACE_DIAMETER);
        }
        drawFaceSansMouth(canvas, i);
        //Draw mouth:
        canvas.setColor(Color.RED);
        canvas.drawArc(X_MOUTH0 + 50*i, Y_MOUTH0 + 30*i,
                              MOUTH_WIDTH, MOUTH_HEIGHT0 + 3*i,
                              MOUTH_START_ANGLE, MOUTH_DEGREES_SHOWN);
    }
    //i == 5

    //Draw kissing face:
    drawFaceSansMouth(canvas, i);
    //Draw mouth in shape of a kiss:
    canvas.setColor(Color.RED);
    canvas.fillOval(X_MOUTH0 + 50*i + 10, Y_MOUTH0 + 30*i,
                          MOUTH_WIDTH - 20, MOUTH_WIDTH - 20);
    //Add text:
    canvas.setColor(Color.BLACK);
    canvas.drawString("Kiss, Kiss.",
                  X_FACE0 + 50*i + FACE_DIAMETER, Y_FACE0 + 30*i);

    //Draw blushing face:
    i++;
    //Draw face circle:
    canvas.setColor(Color.PINK);
    canvas.fillOval(X_FACE0 + 50*i, Y_FACE0 + 30*i,
                          FACE_DIAMETER, FACE_DIAMETER);
```

图4-24　（续）

```
            drawFaceSansMouth(canvas, i);
            //Draw mouth:
            canvas.setColor(Color.RED);
            canvas.drawArc(X_MOUTH0 + 50*i, Y_MOUTH0 + 30*i, MOUTH_WIDTH,
                                   MOUTH_HEIGHT0 + 3*4,//i == 4 is the smile
                                   MOUTH_START_ANGLE, MOUTH_DEGREES_SHOWN);
            //Add text:
            canvas.setColor(Color.BLACK);
            canvas.drawString("Tee Hee.",
                        X_FACE0 + 50*i + FACE_DIAMETER, Y_FACE0 + 30*i);
        }
    }
```

得到的applet

生成的applet与图3-20中显示的完全相同。

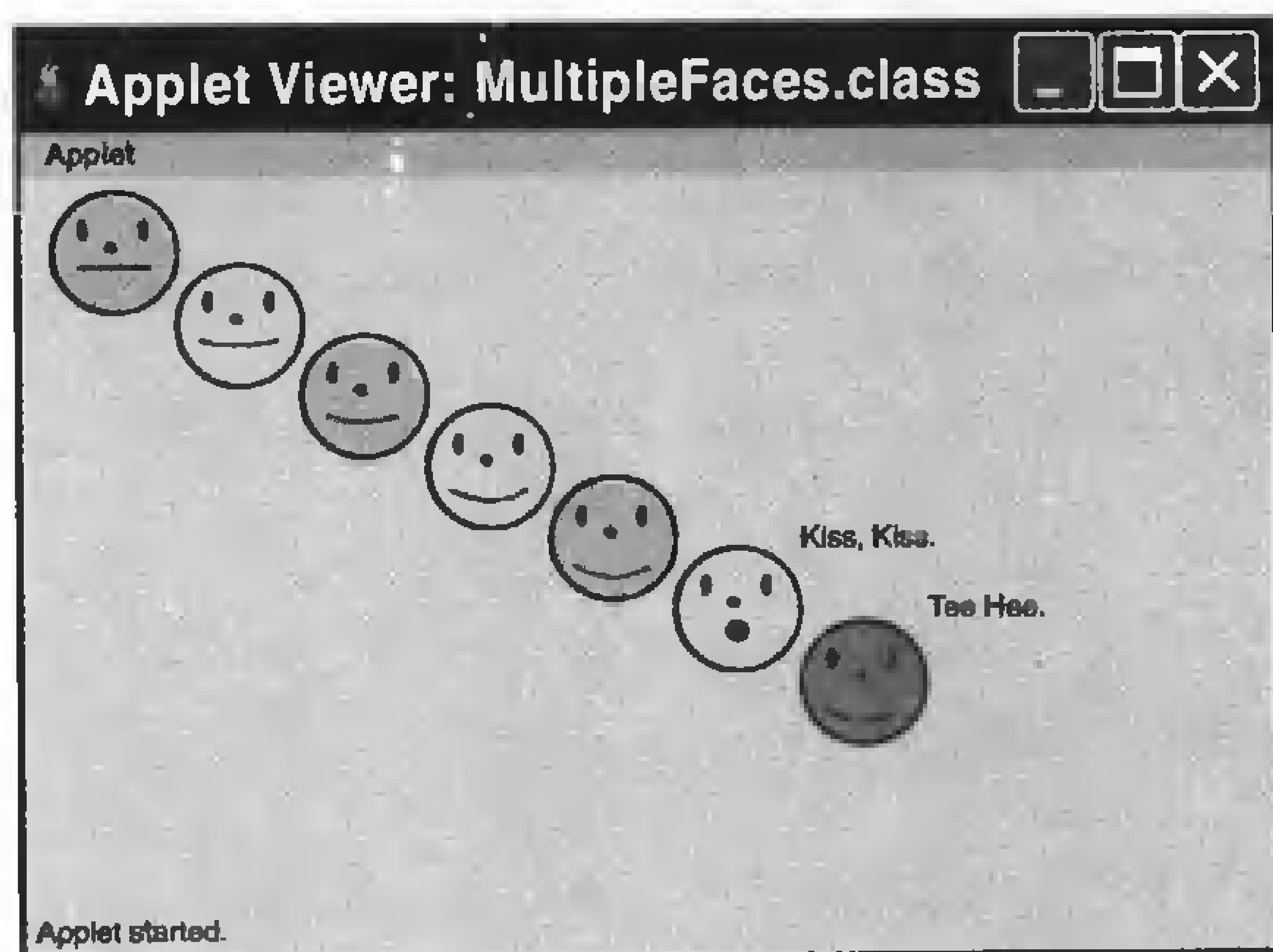


图4-24 （续）

自测题

34. 重新编写图4-24中的drawFaceSansMouth方法，为其添加一个用来表示皮肤颜色的附加参数，并根据此参数对皮肤进行着色。

4.4.2 init方法

到目前为止，每当定义一个applet时，总是会定义paint方法。在编写applet时，还可以定义另一个名为init的常用applet方法。像paint一样，init方法是在运行applet时自动调用的。对applet来说，在某种程度上，init方法是比paint方法更基本的方法。applet中init方法的作用与应用程序中main方法的作用类似：在运行一个Java程序时，程序类中的main方法会被自动调用；运行一个applet时，比如用一个applet查看程序运行applet时，就会自动调用applet类的init方法。每次定义applet类时，通常都要定义init方法。paint方法只是用于绘图（及类似工作）的。applet中所有其他动作都在init方法中运行，或者至少是从init方法中启动的。4.4.3节给出了一个带有init方法的applet示例。

applet类可以既包含init方法的定义，又包含paint方法的定义。还可以包含任何其他想

包含的方法定义，如图4-24中的drawFaceSansMouth方法。但是，方法init和paint是比较特殊的，因为它们是在applet运行时自动调用的。

4.4.3　向applet中添加标签

标签（label）不仅仅是一个带引号的字符串，对它的处理方式与很多稍后要介绍的applet组件（如按钮）的处理方式是一样的。因此，标签就是另一种向applet添加文本的方式，而且它们还提供了一种了解如何向applet添加内容的途径，最终，除了标签之外，还会向applet添加很多其他内容。

图4-25中的applet显示了文本"Hello out there!"，但这个applet没有用drawString来创建所要显示的文本，而是使用了标签。首先要注意，向applet添加组件，比如添加一个标签时，使用的不是paint方法，而是init方法。运行一个applet时，方法paint和方法init都会被自动调用，但你可以在方法paint和init中做不同的事情：用方法paint来画图，用方法drawString来书写文本，用方法init向applet添加一些东西。在本节中，只在方法init中添加了标签。第5章将用init方法向applet中添加按钮。一个applet定义可能会包含一个paint方法、一个init方法或者两者都包含。

```
import javax.swing.*;
import java.awt.*;

/**
 An applet with text using a label.
*/
public class LabelDemo extends JApplet
{
    public void init()
    {
        Container contentPane = getContentPane();
        contentPane.setBackground(Color.WHITE);

        //Create labels:
        JLabel label1 = new JLabel("Hello ");
        JLabel label2 = new JLabel("out there!");

        //Add labels:
        contentPane.setLayout(new FlowLayout());
        contentPane.add(label1);
        contentPane.add(label2);
    }
}
```

得到的applet



图4-25　向applet添加标签

在开始讨论图4-25中代码的主要细节之前，先解释一下图中applet中用到的一些简单但陌生的部分。首先注意下列来自那个applet的代码行：

```
Container contentPane = getContentPane();
```

方法返回的是applet内容面板（content pane）。可以把内容面板当成applet的内部区域。向applet添加组件时，就是将组件添加到了applet的内容面板中。applet的内容面板就是一个Container类型的对象。Container是Java预定义好的一种类型，这里主要用这个类型来声明变量，以命名applet的内容面板。我们讨论的行代码等效于下面两行代码：

```
Container contentPane;
contentPane = getContentPane();
```

其中第一行代码将标识符contentPane声明为一个Container类型的变量。第二行代码获取了applet的内容面板，并将其命名为contentPane。可以用任何（除了关键字之外的）其他标识符来取代contentPane。

常见问题：方法paint和init之间有什么区别？

运行applet时，paint和init方法都会被自动调用，但可以在这两个方法中做不同的事情。可以用方法paint来画图，用方法drawString来书写文本。用方法init向applet添加标签和按钮等。一个applet中可以有一个paint方法，或者一个init方法，或者两者都有。尽管定义一个既不包含paint方法，又不包含init方法的applet是合法的，但这种情况是非常少见的。除非applet的定义中至少包含了这两个方法之一，否则，这个applet很难起到什么作用。

一旦给内容面板命名，比如contentPane，那么，每次想对applet的内容面板做些什么的时候，都可以用这个名字来代替getContentPane方法。例如，图4-25中的下列代码行就将内容面板的颜色设置为白色：

```
contentPane.setBackground(Color.WHITE);
```

快速参考：setBackground方法

可以用方法setBackground为applet的内容面板以及大多数其他组件设置颜色。（如果没有给出颜色，方法就会将内容面板或其他组件设置为默认颜色。）

举例：

```
Container contentPaneOfApplet = getContentPane();
contentPaneOfApplet.setBackground(Color.PINK);
```

在图4-25中，向applet添加了两个标签。添加一个带有所有文本的标签可以获得同样的结果，但我们想要一个向applet添加多个组件的实例。通常会向一个applet中添加多个项。向一个applet添加多个组件时，需要说明这些组件如何排列。在图4-25中，是用如下方式实现的：

```
contentPane.setLayout(new FlowLayout());
```

这条语句说明，向contentPane添加的组件是按照它们的添加顺序从左至右排列的。最终会对这行代码进行详细的解释，但现在只要注意到这行代码的实际效果即可，即了解添加到contentPane中的组件是按它们的添加顺序从左至右排列的。

创建标签及把它们添加到applet内容面板是分两步来执行的：创建标签，然后将标签添加到内容面板。可以用如下列所示图4-25中的代码行创建一个标签：

```
JLabel label1 = new JLabel("Hello");
```

如图4-25中的下列代码行所示，标签是用方法add添加到applet的内容面板中去的：

```
contentPane.add(label1);
```

常见问题：为什么要用标签？用drawString为什么不行？

简单地使用（paint方法中的）drawString方法就可以获得相同效果时，在applet中使用标签好像需要做很多工作。到目前为止，确实是这样的，但很快你就会看到，我们可以用标签做一些特殊的事情。而且，一旦学会了如何添加标签，就可以很容易地在applet中添加很多其他类型的组件，如按钮，因为将它们添加到applet中去的方法和添加标签的方法是一样的。从5.8节开始，你就会看到所有这些内容了。

快速参考：向applet添加标签

用applet的内容面板作为调用对象来使用方法add，就可以向这个applet添加标签了。这个调用出现在applet的init方法中。

语法：

Container *Name_For_Content_Pane* = getContentPane();
    ...
JLabel *Name_Of_Label* = new JLabel(*Quoted_String*);
    ...
Name_For_Content_Pane.setLayout(new FlowLayout());
    ...
Name_For_Content_Pane.add(*Name_Of_Label*);

举例：

参见图4-25。

自测题

35. 如果用下列代码取代图4-25中的init方法，会对图4-25中的applet产生什么影响？（下列代码只是用标识符inside取代了contentPane。）

```
public void init()
{
    Container inside = getContentPane();
    inside.setBackground(Color.WHITE);
    //Create labels:
    JLabel label1 = new JLabel("Hello ");
    JLabel label2 = new JLabel("out there!");
    //Add labels:
    inside.setLayout(new FlowLayout());
    inside.add(label1);
    inside.add(label2);
}
```

36. 如果我们用代码

```
getContentPane().add(label1);
```

取代图4-25中的下列代码：

```
contentPane.add(label1);
```

（除了风格上的变化之外）会产生什么影响？

小结

- 类中拥有一些存储数据的实例变量和一些执行动作的方法。
- 类中所有的实例变量都应该声明为private。将其声明为private后，除非是在同一个类的方法定义内部，否则就不能通过名字来访问它们了。
- 封装意味着将数据和动作组合到一个单一的条目（在目前这种情况下，就是类对象）中，而且实现的细节都被隐藏了。将所有的实例变量都声明为private是封装过程的一部分。
- 类类型的变量是引用变量。这就意味着类类型的变量中含有的是存储它所命名对象的内存地址。
- 有两种类型的方法：会返回值的方法和void方法。
- 方法的参数可以是基本类型的，也可以是类类型的，但这两种类型参数的表现有所不同。
- 基本类型的参数是一个局部变量，会在方法调用时被初始化为相应实参的值。这种用实参来替代形参的方法称为按值调用机制。
- 类类型的参数会成为方法调用中相应实参的另一个名字。这样，对参数所做的任何修改都会作用到相应的实参上。
- 对一个类的对象使用运算符=和==时，这些运算符的表现与将其用于基本类型时的表现不同。
- 通常都要为你定义的类定义一个equals方法。
- applet的绘图方法通常都在paint方法中使用。大多数其他applet指令都在init方法中使用。[①]
- applet中paint方法的参数是Graphics类型的。我们用到的applet绘图方法都是Graphics类中的方法。[①]

✓自测题答案

1.
```
SpeciesFirstTry speciesOfTheYear = new SpeciesFirstTry();
System.out.println("Enter data for Species of the Year:");
speciesOfTheYear.readInput();
```
2.
```
dilbert.readInput();
```
3.
```
import java.util.*;
public class SpeciesFirstTry
{
   public String name;
   public int number;
   public int population;
   public double growthRate;
   public void readInput()
   {
        Scanner keyboard = new Scanner(System.in);
        System.out.println("What is the species' name?");
        name = keyboard.nextLine();
        System.out.println("What is the species' number?");
        number = keyboard.nextInt();
        while (number < 0)
        {
             System.out.println("Number cannot be negative.");
             System.out.println("Reenter number:");
             number = keyboard.nextInt();
```

①在4.4节中介绍的内容。

```
}
System.out.println("What is the population of the species?");
population = keyboard.nextInt();
while (population < 0)
{
     System.out.println("Population cannot be negative.");
     System.out.println("Reenter population:");
     population = keyboard.nextInt();
}
System.out.println("Enter growth rate (percent increase per year):");
growthRate = keyboard.nextDouble();
}
public void writeOutput()
{
    System.out.println("Name = " + name);
    System.out.println("Number = " + number);
    System.out.println("Population = " + population);
    System.out.println("Growth rate= " + growthRate + "%");
}
public int populationIn10()
<这个方法没有变化。>
}
```

4.
```
public int femalePopulation()
(
    return (population/2 + population%2);
}
public int malePopulation()
{
    return population/2;
}
```

5.
```
public void writeOutput()
{
    System.out.println("Name = " + this.name);
    System.out.println("Population = " + this.population);
    System.out.println("Growth rate = " + this.growthRate + "%");
}
```

6.
```
import java.util.*;
public void readInput()
{
    Scanner keyboard = new Scanner(System.in);
    System.out.println("What is the species' name?");
    this.name = keyboard.nextLine();
    System.out.println("What is the population of the species?");
    this.population = keyboard.nextInt();
    while (this.population < 0)
    {
    System.out.println("Population cannot be negative.");
    System.out.println("Reenter population:");
    this.population = keyboard.nextInt();
    }
```

```
    System.out.println(
        "Enter growth rate (percent increase per year):");
    this.growthRate = keyboard.nextDouble();
}
```

7.
```
public int populationIn10()
{
    double populationAmount = this.population;
    int count = 10;
    while ((count > 0) && (populationAmount > 0))
    {
        populationAmount = (populationAmount +
                (this.growthRate/100) * populationAmount);
        count--;
    }
    if (populationAmount > 0)
        return (int)populationAmount;
    else
        return 0;
}
```

8. 表达式(int)是一种强制类型转换。因为方法头部说明了返回值的类型为int，在返回populationAmount值之前必须将其从double类型改成int类型，所以要用到强制类型转换。

9. 在本书中，术语*parameter*和*formal parameter*的含义相同。

10.
```
public double density(double area)
{
    return population/area;
}
```

11.
```
public void fixPopulation(double area)
{
    population = (int)(2*area);
}
```

12.
```
public void changePopulation(double area, int numberPerMile)
{
    population = (int)(numberPerMile*area);
}
```

13. 因为在类定义中实例变量被标记为private，除了在SpeciesFourthTry类的方法定义内部，无法直接对其进行访问，所以不能使用替换代码。

14.
```
/**
Precondition: Calling object's population and growth
rate have been given values.
Postcondition: Calling object's population was updated to
reflect one year's change.
Other data values are unchanged.
*/
```

15. 断言是一条用来描述程序状态的语句。断言可以为真也可以为假，但如果程序没有出错就应该为真。前置条件和后置条件语句是断言的实例。下面是另一个分两次给出的断言实例：一次是作为注释给出的，一次是作为断言检测给出的。

```
//(timeLeft > 30) && (points < 10)
assert (timeLeft > 30) && (points < 10);
```

16.
```
if (balance <= 0) //if (balance > 0) is false.
{
```

```
    System.out.println("Assertion (balance > 0) failed.");
    System.out.println("Aborting program.");
    System.exit(0);
}
```

17. 访问方法是从一个或多个私有实例变量中读取或返回数据的公有方法。(访问方法的名字通常以get开头。) 设置方法是对存储在一个或多个私有实例变量中的数据进行修改的公有方法。(设置方法的名字通常以set开头。)

18.
```
import java.util.*;
public class Person
{
    private String name;
    private int age;
    public void readInput()
    {
        Scanner keyboard = new Scanner(System.in);
        System.out.println("What is the person's name?");
        name = keyboard.nextLine();
        System.out.println("What is the person's age?");
        age = keyboard.nextInt();
        while (age < 0)
        {
            System.out.println("Age cannot be negative.");
            System.out.println("Reenter age:");
            age = keyboard.nextInt();
        }
    }
    public void writeOutput()
    {
        System.out.println("Name = " + name);
        System.out.println("Age = " + age);
    }
    public void set(String newName, int newAge)
    {
        name = newName;
        if (newAge >= 0)
            age = newAge;
        else
        {
            System.out.println("ERROR: Used a negative age.");
            System.exit(0);
        }
    }
    public String getName()
    {
        return name;
    }
    public int getAge()
    {
        return age;
    }
}
```

19. 可以清晰地划分为用户接口和实现的类定义就是封装良好的类定义。使用这种类的程序员只需要了解用户接口，而不需要关心实现的细节。

20. 实例变量总是应该标记为private。
21. 如果一个方法是仅在其他方法定义中使用的辅助方法，就应该将其标记为private。
22. 不，是实现的一部分。
23. 不，是实现的一部分。
24. 引用类型是一种在其变量中包含了引用（即包含了内存地址）而不是对象实际值的类型。类类型是引用类型。（除了类类型之外还有其他引用类型，但要到本书稍后才会见到。）基本类型不是引用类型。
25. 通常，在对两个方法进行测试，看其是否"相等"时，要使用方法equals。只有在想看两个对象是否位于内存中同一个位置时，才能使用==，一般来说你不太可能做这样的测试。
26. 在比较两个int这样的基本类型的对象，看它们是否"相等"时，要使用==。实际上，对基本类型来说，通常没有equals方法。
27.
```
/**
  Precondition: The calling object and the argument
  otherSpecies both have values for their growth rates.
  Returns true if the growth rate of the calling object is
  greater than the growth rate of otherSpecies;
  otherwise, returns false.
  */
  public boolean largerGrowthRateThan(Species otherSpecies)
  {
  return (growthRate > otherSpecies.growthRate);
}
```
28. 变量s1和s2是Species类型对象的名字，但这个程序并没有创建任何以它们命名的对象。它们只是名字，还不是对象。程序应该这样开始：
```
public class SpeciesEqualsDemo
{
    public static void main(String[] args)
    {
           Species s1 = new Species(), s2 = new Species();
           <其余的代码都是对的。>
```
29. 最大的区别在于方法是如何对对应于不同类型参数的实参进行处理的。方法无法改变作为方法实参的基本类型变量的值，但可以改变作为方法实参的类类型实例变量的值。
30.
```
Name = Klingon ox
Population = 10
Growth rate = 15.0%
n = 0
```
31. 除了增加了equals方法之外，类定义与以前一样。下面是两种可能的equals定义：第一种相当于说一个人在某个年龄时和这个人在另一个（可能不同的）年龄时是一样的，第二种相当于说一个人在某个年龄时和他在另一个年龄时是不同的。
```
public boolean equals(Person otherObject)
{
    return (this.name.equalsIgnoreCase(otherObject.name));
}
public boolean equals(Person otherObject)
{
    return ((this.name.equalsIgnoreCase(otherObject.name))
           && (this.age == otherObject.age));
}
```
如果省略所有的this和跟在this后面的点，也是对的。

32. 不会有什么影响。applet会表现得和以前完全一样。因为canvas只是一个参数，可以用任何其他（非关键字的）标识符来取代canvas。
33. applet运行时，会自动调用方法paint。
34.
```
private void drawFaceSansMouth(Graphics g, int i, Color skinColor)
{
    g.setColor(skinColor);
    g.fillOval(X_FACE0 + 50*i, Y_FACE0 + 30*i,
                         FACE_DIAMETER, FACE_DIAMETER);
    g.setColor(Color.BLACK);
    g.drawOval(X_FACE0 + 50*i, Y_FACE0 + 30*i,
                         FACE_DIAMETER, FACE_DIAMETER);
    //Draw eyes:
    g.setColor(Color.BLUE);
    g.fillOval(X_RIGHT_EYE0 + 50*i, Y_RIGHT_EYE0 + 30*i,
                              EYE_WIDTH, EYE_HEIGHT);
    g.fillOval(X_LEFT_EYE0 + 50*i, Y_LEFT_EYE0 + 30*i,
                              EYE_WIDTH, EYE_HEIGHT);
    //Draw nose:
    g.setColor(Color.BLACK);
    g.fillOval(X_NOSE0 + 50*i, Y_NOSE0 + 30*i,
                            NOSE_DIAMETER, NOSE_DIAMETER);
}
```
35. 没什么影响。applet看起来和以前完全一样。
36. 没什么影响。applet看起来和以前完全一样。

● 编程项目

1. 编写一个程序来回答下面这样的问题：假设Klingon ox物种的种群规模为100，增长率为15%，elephant物种的种群规模为10，增长率为35%。要过多少年elephant物种的种群规模才能超过Klingon ox的种群规模？使用图4-19中的Species类。程序要询问与这两个物种有关的数据，并以开始时种群规模较小的物种数量超过开始时种群规模较大的物种数量所需的年数作为响应。可以以任意顺序输入两个物种。注意，种群规模较小的物种的数量可能永远也无法超过另一物种。在这种情况下，程序应该输出一条适当的信息来说明这种情况。
2. 定义一个名为Counter的类。用这个类的对象来计数，使其记录一个非负整数值。要包含将计数器设置为0、将计数器加1、将计数器减1的方法。要确保没有任何方法会使计数器的值变成负的。还要包含一个返回当前计数值的访问方法，以及一个将计数输出到屏幕上去的方法。没有输入方法。唯一可以设置计数器的方法就是把它设置为0的方法。编写一个程序来测试类定义。（提示：只需要使用一个实例变量。）
3. 为具有下列分级策略的类编写一个定级程序：
   (a) 有两次小测验，每次都以10点为基础评分。
   (b) 有一次期中考试和一次期末考试，每次都以100为基础评分。
   (c) 期末考试占分数的50%，期中考试占25%，两次小测验合在一起占25%。（不要忘记对小测验的分数进行标准化。在将其计入平均值之前，要先将其转换成百分数。）

   90或90以上的分数都为A，80或80以上（但小于90）的分数都为B，70或70以上（但小于80）的分数都为C，60或60以上（但小于70）的分数都为D，小于60的分数都为F。程序应该读入学生的分数，并输出学生的记录，其中应该包括两次小测验分数和两次考试分数，以及学生整门课程的总体数字分数

和最后的字母等级。

为学生记录定义并使用一个类。这个类中应该包含用于小测验、期中考试、期末考试及总体数字分数的实例变量。总体数字分数是一个取值范围为0~100的数字，用来表示学生成绩的加权平均值。这个类应该有一些输入和输出方法。输入方法不应该询问最终的数字等级，也不应该询问最终的字母等级。类中应该有一些用来计算总体数字分数和最终字母等级的方法。最后两个方法应该是void方法，用来对适当的实例变量进行设置。记住，一个方法可以调用另一个方法。如果愿意，可以定义一个方法来设置总体数字分数和最终字母等级，但如果这样做，就要使用一个辅助方法。程序中应该用到我们讨论过的所有方法。无论程序中是否用到，类中都应该有一个合理的访问方法和设置方法集。如果愿意，也可以添加其他方法。

4.向自测题18中的Person类添加一些仅用来设置Person的名字属性、仅用来设置Person的年龄属性、测试两个Person是否相等（具有相同的名字和年龄），测试两个Person是否具有相同的名字、测试两个Person是否具有相同的年龄、测试一个Person是否比另一个Person老、测试一个Person是否比另一个Person年轻的方法。编写一个对每种方法都进行了演示的驱动（测试）程序，至少要用一种为真和一种为假的情形来测试每种被测方法。

5. 创建一个描绘了一个等级分布（A、B、C、D、F级中分数的数量）图的类，这个类会在水平方向上打印一些行，每行上的星号数都对应于每个等级所占的百分比。编写一些用来设置每个字母等级数量的方法；读取每个字母等级数量的方法；返回等级总数的方法；返回以一个0~100（包含0和100在内）的整数表示的每个字母等级所占百分比的方法；以及画图的方法。对其进行设置，使50个星号对应于100%（每个星号对应于2%），在水平轴上要有一个标尺，用来指示0~100%之间每10个百分点的增长，并用每行的字母等级来标识这一行。例如，如果有1个A，4个B，6个C，2个D和1个F，那么，等级总数就是14，A所占百分比为7，B所占百分比为29，C所占百分比为43，D所占百分比为14，F所占百分比为7。A行应该包含4颗星（7/50，舍入到最近的整数），B行14颗，C行21颗，D行7颗，F行4颗，因此图形看起来应该如图4-26所示。

```
0    10   20   30   40   50   60   70   80   90   100%
|    |    |    |    |    |    |    |    |    |    |
**************************************************
**** A
************** B
********************* C
******* D
**** F
```

图4-26 等级分布图

6. 编写一个使用了（图4-11中）Purchase类的程序来设置下列价格：

- 橘子：10个2.99美元。
- 鸡蛋：12个1.69美元。
- 苹果：3个1.00美元。
- 西瓜：每个4.39美元。
- 硬面包圈：6个3.50美元。

然后，计算下列账单总额以及各个项的账单小计：

- 2打橘子。
- 3打鸡蛋。
- 20个苹果。
- 2个西瓜。

• 1打硬面包圈。

7. 编写一个程序来回答以下问题：假设物种Klingon ox的种群规模为100，增长率为15%，而且居住在一个1500平方英里的区域中。要使种群密度超过每平方英里1个需要花费多长时间？使用添加了自测题10中的`density`方法的、图4-19中的`Species`类。
8. （这个项目要求学习过4.4节的内容。）编写一个applet来显示一只公牛眼睛，眼睛与图4-27所示类似。使用一个将显示圆圈作为其子任务的方法。

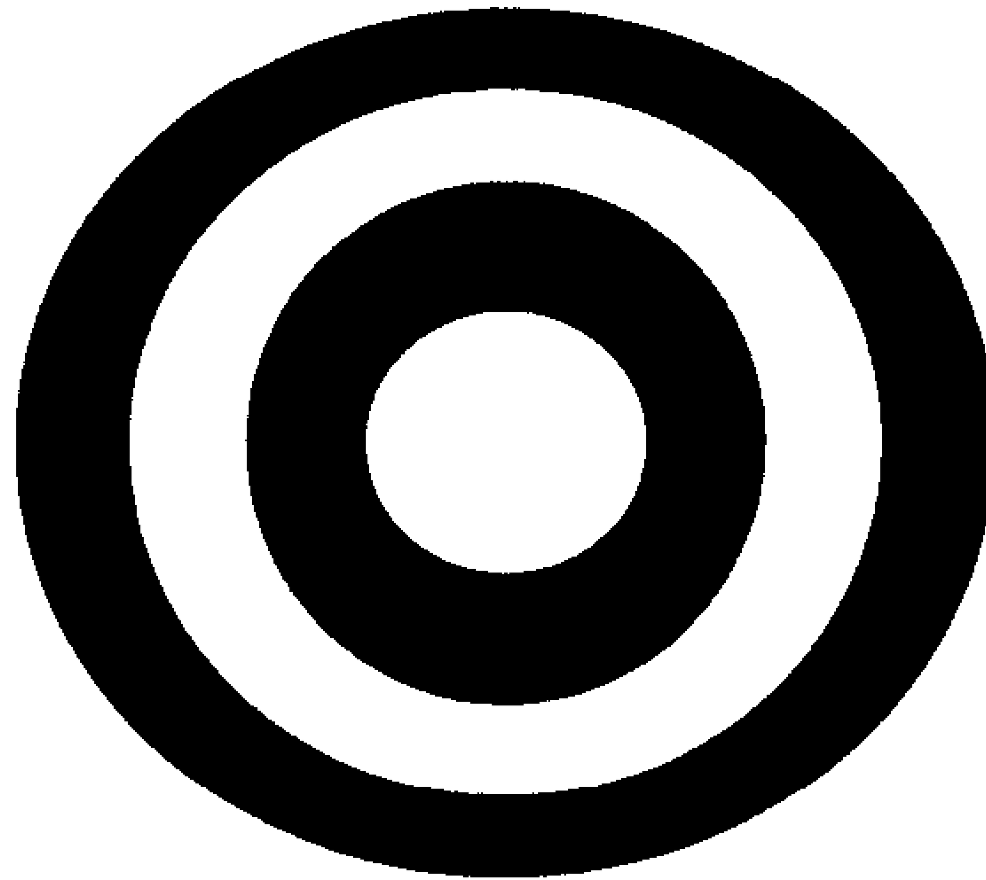


图4-27 公牛眼睛

第5章

对象与方法

一个旅行者在纽约市的街上拦住一位老年绅士，问他："先生，请问你能告诉我怎么才能到达卡内基大厅吗？"

"练习，练习，练习，"老绅士回答。

——一个很老的笑话

本章将继续介绍如何定义和使用类及其方法。本章讨论**重载**（overloading）技术，这种技术可以使你在同一个类中拥有两个或多个具有相同名字的方法；介绍如何定义静态方法（静态方法是可以用类名而不是对象名调用的方法）；介绍构造器（构造器是用来对新对象进行自动初始化的方法）；还讨论了包（包是可以方便地在其他类定义中使用的类库）；另外，还介绍了很多程序设计技巧，以帮助你编写出更好的方法定义。

选读的5.8节，介绍如何向applet中添加按钮和图标（一些小图片），还介绍了**事件驱动编程**（event driven programming），这是在applet和其他图形程序编程中常用的一种技巧。

目标

- 熟悉更多与类及对象的编程有关的技巧。
- 了解静态方法和静态变量。
- 了解对方法名的重载。
- 学会在Java中定义构造器的方法。
- 了解Java中的包和输入语句。
- 学会使用自顶向下的设计技巧来设计方法。
- 学习大量用来测试方法的技巧，包括存根方法和驱动程序的使用。
- 选读，学习向applet中添加按钮和图标。
- 选读，学习**事件驱动编程**技术的基础知识。

预备知识

在阅读本章之前必须学习第4章的内容。本章的某些小节可以在阅读一些后继章节之后再学习。

5.6节介绍了使用类类型实例变量的一些细节。严格来讲，本书的其他内容都不需要这一节的支持，所以可以推迟学习这一节。但5.6节对一些基本问题的讨论对学习某些内容还是很有用的。

学习5.7节关于包的内容时，需要对目录（文件夹）和路径变量有所了解。目录（文件夹）

和路径变量都不是Java的主题，而是操作系统中的主题，具体的细节取决于你使用的是什么操作系统。本书的其他内容都不需要5.7节的支持，所以可以在准备好以后再学习5.7节。

5.1　用方法编程

19世纪最伟大的发明就是发明方法的发明。

——艾·诺·怀特海，《科学与现代世界》

本节会讨论很多可以在设计和测试方法时使用的基本技巧。

在方法中调用方法

一个方法的主体中可能会包含对另一个方法的调用。这种类型的方法调用与出现在程序main部分的方法调用完全一样。但是，如果所有的方法都在同一个类中，对其进行调用时通常无需写出调用对象了。

图5-1中包含了一个名为Oracle的类定义。这个类的dialog方法构建了一个与用户之间的对话，用来回答用户提出的一系列单行问题。注意，在dialog方法的定义内部，有一个对方法answerOne的调用，这个方法也是同一个Oracle类中的方法。查看一下方法answerOne的定义，就会发现它又包含了对另外两个方法的调用，即seekAdvice和update，这两个方法也位于同一个Oracle类中。

首先考虑一下dialog方法定义中对方法answerOne的调用。图5-2中包含了一个演示程序，这个程序创建了一个名为delphi的Oracle类的对象，并用这个对象调用了方法dialog。调用发生时，dialog方法是用对象delphi执行的。dialog方法的定义如下所示：

```
public void dialog()
{
    String ans;
    Scanner keyboard = new Scanner(System.in);
    do
    {
        answerOne();
        System.out.println("Do you wish to ask another question?");
        ans = keyboard.next();
    }while (ans.equalsIgnoreCase("yes"));
    System.out.println("The oracle will now rest.");
}
```

注意，answerOne方法前面没有对象和点。它的调用对象就是方法dialog的调用对象。在图5-2所示的程序中，dialog方法是按照下列方式由对象delphi调用的：

```
delphi.dialog();
```

因此，对dialog()的这次调用来说，可以认为dialog定义中的调用

```
answerOne();
```

意味着

```
delphi.answerOne();
```

```
import java.util.*;

public class Oracle
{
    private String oldAnswer = "The answer is in your heart.";
    private String newAnswer;
    private String question;
    public void dialog()
    {
       String ans;
       Scanner keyboard = new Scanner(System.in);
       do
        {
            answerOne();
            System.out.println("Do you wish to ask another question?");
            ans = keyboard.next();
        } while (ans.equalsIgnoreCase("yes"));
        System.out.println("The oracle will now rest.");
    }
    private void answerOne()
    {
        System.out.println("I am the oracle.");
        System.out.println("I will answer any one-line question.");
        System.out.println("What is your question?");
        Scanner keyboard = new Scanner(System.in);
        question = keyboard.nextLine();
        seekAdvice();
        System.out.println("You asked the question:");
        System.out.println(question);
        System.out.println("Now, here is my answer:");
        System.out.println(oldAnswer);
        update();
    }
    private void seekAdvice()
    {
        System.out.println("Hmm, I need some help on that.");
        System.out.println("Please give me one line of advice.");
        Scanner keyboard = new Scanner(System.in);
        newAnswer = keyboard.nextLine();
        System.out.println("Thank you. That helped a lot.");
    }

    private void update()
    {
        oldAnswer = newAnswer;
    }
}
```

图5-1 调用其他方法的方法

编写dialog这样的方法定义时，并不知道调用对象名。调用对象在不同的情况下可能是不同的，比如，在一个程序中是

```
delphi.dialog();
```

在另一个程序中可能就是

```
myObject.dialog();
```

因为不知道（实际上也无法知道）调用对象的名字，所以省略了它。因此，在图5-1所示的Oracle类定义中，在dialog方法的定义中写下

```
answerOne();
```

时，表示的是

The_Calling_Object.answerOne();

```
public class OracleDemo
{
    public static void main(String[] args)
    {
        Oracle delphi = new Oracle();
        delphi.dialog();
    }
}
```

屏幕对话示例

```
I am the oracle.
I will answer any one-line question.
What is your question?
What time is it?
Hmm, I need some help on that.
Please give me one line of advice.
Seek and ye shall find the answer.
Thank you. That helped a lot.
You asked the question:
What time is it?
Now, here is my answer:
The answer is in your heart.
Do you wish to ask another question?
yes
I am the oracle.
I will answer any one-line question.
What is your question?
What is the meaning of life?
Hmm, I need some help on that.
Please give me one line of advice.
Ask the car guys.
Thank you. That helped a lot.
You asked the question:
What is the meaning of life?
Now, here is my answer:
Seek and ye shall find the answer.
Do you wish to ask another question?
no
The oracle will now rest.
```

图5-2　Oracle演示程序

因为参数this就表示*The_Calling_Object*，所以可以用参数this来命名调用对象。因此，

下面两次对answerOne的调用是等效的：

```
answerOne();
```

与

```
this.answerOne();
```

引用同一个类中的方法时省略this和点与在实例变量中的用法相同。这种省略调用对象和点的方式只适用于同一个类中的方法。如果在一个类的方法定义中调用另一个类的方法，就必须包含对象和点。①

还要注意，只有在可以用this参数表示调用对象时，才能采用这种省略调用对象的方法。这是对能否省略调用对象和点的一种简单可靠的测试方法。如果调用对象是在方法定义内部声明、并用new创建的对象，那么，在那个方法定义内部，就必须包含对象名和点。

下面继续介绍Oracle类中的方法定义。你可以使用调用其他方法的方法，而这个方法所调用的方法又可以接着调用其他的方法。在Oracle类中，方法answerOne的定义中包含了对方法seekAdvice和update的调用，这两个方法也位于同一个Oracle类中。正如刚才讨论的那样，这两个名为seekAdvice和update的方法的前面是没有对象和点的。

下面是图5-2所示程序中的调用：

```
delphi.dialog();
```

dialog的定义中包含了调用

```
answerOne();
```

这个调用等效于

```
this.answerOne();
```

由于调用对象为delphi，所以这个调用也等效于

```
delphi.answerOne();
```

answerOne的定义中包含了调用

```
seekAdvice();
```

和

```
update();
```

这两个调用等效于

```
this.seekAdvice();
```

和

```
this.update();
```

由于调用对象还是delphi，所以它们也等效于

```
delphi.seekAdvice();
```

和

```
delphi.update();
```

可以用方法来调用那些调用了方法的方法，这种方法调用可以进行任意次。调用的细节通常都是按照我们在这里描述的方式来处理的。

记住：省略调用对象

当方法调用中的调用对象为参数this时，可以将this和点省略。

① 这条在一个类定义中调用另一个类的方法的规则不适用于5.2节中讨论的静态方法，但适用于到目前为止已经讨论过的所有类型的方法。

举例：

下列两个代码段是等效的：

```
public void answerOne()
{
        .
        .
        .
    this.seekAdvice();
        .
        .
        .
    this.update();
}
```

和

```
public void answerOne()
{
        .
        .
        .
seekAdvice();
        .
        .
        .
update();
}
```

● 编程提示：将辅助方法设为私有的

图5-1中的方法answerOne、seekAdvice和update都被标记为private而不是public。如果方法被标记为private，就只能在同一个类的其他方法定义中使用。因此，在其他类或程序中，下列对私有方法answerOne的调用将是非法的，并且会产生一条编译器错误消息：

```
Oracle myOracle = new Oracle();
myOracle.answerOne();  //Invalid: answerOne is private.
```

而下列对公有方法dialog的调用则是完全合法的：

```
myOracle.dialog(); //Valid.
```

将方法answerOne、seekAdvice和update标记为private是因为它们只是一些辅助方法。Oracle类的用户不能使用这些方法。它们只能在dialog方法的定义中使用。这就意味着方法answerOne、seekAdvice和update是类实现的一部分，而不是类用户接口的一部分。正如4.2.4节中讨论的，将类的实现部分设置为私有是一种很好的编程习惯。

■ Java提示：使编译器工作顺畅

编译器会试着查看程序，以确保你完成了一些必要的任务，如初始化变量，或者在返回值的方法定义中包含一条return语句。有时你可能会发现，编译器要求你做的事情是：要么是确定你已经做了，要么是确定不用做。在这种情况下，不必与编译器争执。只要改动一些内容，使编译器不再“抱怨”即可。首先，要查看程序以确保编译器的提示是不是正确的。通常情况下编译器都是正确的。如果在代码中找不到真正的错误，可以修改代码使其更明显地表明已经完成了编译器要求

的工作。

例如，如果按下列方式声明了一个变量line：

```
String line;
```

而编译器坚持认为必须对变量line进行初始化，就可以将声明改成下面的形式：

```
String line = null;
```

常量值null是一个特殊常量，可以用它为任意类类型的变量赋值。

再如，假设有一个会返回int类型值的方法，而且方法的定义是以如下形式结束的：

```
if (something > somethingElse)
    return something;
else
    return somethingElse;
```

在这种情况下，每次计算都会以一条return语句结束。如果编译器仍然指出，需要一条return语句，可以将最后这条if-else语句改成如下形式：

```
int answer;
if (something > somethingElse)
    answer = something;
else
    answer = somethingElse;
return answer;
```

快速参考：null

null是一个特殊常量，可以用来为任意类类型的变量赋值。常量null不是一个对象，而是一种对象地址占位符。因为它像一个地址一样，所以在测试一个变量是否等于null时，要使用==和!=，而不是equals方法。

▲ 易犯错误：(Null）指针异常

如果编译器请你初始化一个类变量，通常都可以将这个变量初始化为null。但是，null不是一个对象，因此不能用被初始化为null的变量来调用对象。如果试图这么做，会得到一条错误消息"Null指针异常"。例如，如果将下列代码放在一个程序里，就会产生"Null指针异常"：

```
Species specialSpecies = null;
System.out.println("Enter data on special species:");
specialSpecies.readInput();
```

Species类是在图4-19中定义的。但是，没必要去看那个类定义。无论Species类是如何定义的，这3行代码都会产生一条错误消息。更正这个问题的方法是，按下列方法用new来创建一个Species类型的对象：

```
Species specialSpecies = new Species();
System.out.println("Enter data on special species:");
specialSpecies.readInput();
```

只有确信将变量用作某个方法的调用对象之前，程序代码应该用new（或其他的方法）为这个类变量分配一个对象时，才能用null对类变量进行初始化。这就意味着，只有在概念上不需要进行初始化时，才能用null来初始化一个类变量。但是，编译器有时会在不需要进行初始化时坚持要求进行初始化。

随着编写的代码越来越多，你可能会遇到其他会产生"Null指针异常"消息的情况。在这些情况下，可以查看有没有未初始化的类变量。 △

自测题

1. 可以在一个方法的定义中调用位于同一个类中的另一个方法吗?
2. 假设将图5-1所示的answerOne方法定义中的代码行

```
seekAdvice();
```

改成了

```
this.seekAdvice();
```

这样做会有什么影响?

3. 以下列代码（图4-19中定义的Species类）开头的程序会有什么问题?

```
public class SpeciesDemo
{
    public static void main(String[] args)
    {
        Species s1 = null;
        Species s2 = null;

        s1.set("Klingon Ox", 10, 15);
        s2.set("Naked Mole Rat", 10000, 25);
```

5.2 静态方法与静态变量

……那里什么都没有了。

——格特鲁德·斯泰因，美国女作家

静态方法和静态变量是完全属于类的方法和变量，不需要任何对象。本节将介绍如何定义静态方法和静态变量。

5.2.1 静态方法

有时候要用到一个不需要任何类型的对象的方法。比如，可能需要一个用来计算两个整数中最大值的方法，或者一个用来计算一个数字平方根的方法，或者一个用来将小写字母转换成大写字母的方法。这些方法都没有任何明显的所属对象。在这些情况下，可以将这些方法定义为**静态**（static）的。将一个方法定义为静态方法是在类中实现的，所以这个方法仍然是类的成员，但调用这个方法时可以不使用任何对象。调用一个静态方法时，使用的通常是类名而不是对象名。

例如，图5-3所示是一个名为CircleFirstTry的类定义，在这个类定义中包含了两个静态方法定义。可以按下例所示的形式来调用这些方法：

```
double areaOfCircle = CircleFirstTry.area(12.7);
double circumOfCircle = CircleFirstTry.circumference(12.7);
```

注意，对静态方法来说，类名的作用和调用对象的作用是一样的。（创建一个CircleFirstTry类的对象，并用这个对象来调用方法area或方法circumference也是可以的，但这是一种容易让人迷惑的方式，所以在调用静态方法时，通常会使用类名。）图5-4给出了另一个使用静态方法的实例。

```
/**
Class with static methods to perform calculations on circles.
*/
public class CircleFirstTry
{
    public static final double PI = 3.14159;

    public static double area(double radius)
    {
        return (PI*radius*radius);
    }

    public static double circumference(double radius)
    {
        return (PI*(radius + radius));
    }
}
```

本章稍后将给出这个类的另一版本。

图5-3 静态方法

```
import java.util.*;

public class CircleDemo
{
    public static void main(String[] args)
    {
        double radius;

        System.out.println(
          "Enter the radius of a circle in inches:");
        Scanner keyboard = new Scanner(System.in);
        radius = keyboard.nextDouble();

        System.out.println("A circle of radius "+ radius + " inches");
        System.out.println("has an area of " +
             CircleFirstTry.area(radius) + " square inches,");
        System.out.println("and a circumference of " +
             CircleFirstTry.circumference(radius) + " inches.");
    }
}
```

屏幕对话示例

```
Enter the radius of a circle in inches:
2.3
A circle of radius 2.3 inches
has an area of 16.61901 square inches,
and a circumference of 14.45131 inches.
```

图5-4 静态方法的使用

静态方法的定义和任何其他方法的定义方式一样，但要在方法头部加上关键字static。

在图5-3中CircleFirstTry类所示的例子中，类中是没有实例变量的，但一个既包含实例变量又包含静态方法（及常规的非静态方法）的类是完全合法的。

在静态方法的定义中，不能做任何与调用对象有关的事情，比如访问一个实例变量。这很好理解，因为调用静态方法时可以不使用任何调用对象，这样在调用静态方法时就没有什么实例变量可以引用了。后面的"易犯错误"部分将对此进行更完整的解释。

▲ 易犯错误：在静态方法内调用非静态方法

图5-5所示为用于圆周计算的类定义。这个类设计得很简单，但它既包含静态方法又包含非静态方法，这样就可以看出这两种类型的方法可以进行及不能进行哪些交互了。方法areaDialog说明了从静态方法内部调用非静态方法的唯一一种合法的方式。非静态方法setDiameter和showArea是从静态方法areaDialog内部调用的。下面介绍其中的一些细节。

```
import java.util.*;

public class PlayCircle
{
    public static final double PI = 3.14159;

    private double diameter;

    public void setDiameter(double newDiameter)
    {
        diameter = newDiameter;
    }

    public static double area(double radius)
    {
        return (PI*radius*radius);
    }

    public void showArea()
    {
        System.out.println("Area is " + area(diameter/2));
    }

    public static void areaDialog()
    {
        Scanner keyboard = new Scanner(System.in);
        System.out.println("Enter diameter of circle:");
        double newDiameter = keyboard.nextDouble();
        PlayCircle c = new PlayCircle();
        c.setDiameter(newDiameter);
        c.showArea();
    }
}
```

（作为常量使用的）静态变量。

实例变量。

只有在用new创建了一个c这样的调用对象时，才能在一个静态方法内部调用一个非静态方法。

图5-5 静态方法和非静态方法的混用

如图5-6中的代码行所示，由于areaDialog是一个静态方法，因此调用它时可以只使用类名而不使用任何对象，代码行如下：

```
PlayCircle.areaDialog();
```

在这个调用中，没有与方法areaDialog相关联的对象。因此，如果想在areaDialog的定义内调用非静态方法showArea，就必须创建一个对象，并用那个对象调用方法showArea。这就是方法areaDialog的定义中所做的工作。代码如下：

```
public static void areaDialog()
{
     Scanner keyboard = new Scanner(System.in);
     System.out.println("Enter diameter of circle:");
     double newDiameter = keyboard.nextDouble();
     PlayCircle c = new PlayCircle();
     c.setDiameter(newDiameter);
     c.showArea();
}
```

相反，假设删除了调用对象c，使方法定义的最后一行如下所示：

```
showArea();
```

这条语句等价于

```
this.showArea();
```

如果对方法areaDialog的定义进行这样的修改，并重新编译PlayCircle的类定义，就会收到一个编译器错误。这是因为任何对方法showArea的调用都必须包含一个带有圆的直径数据的调用对象。它无法通过任何其他方式获得直径数据。在areaDialog这样的静态方法定义中，没有可以用于任何目的的实例变量。注意，问题并不在于类中缺少实例变量。类中有一个很合适的、名为diameter的实例变量。问题在于非静态方法showArea是在静态方法areaDialog内部调用的。

```
public class PlayCircleDemo
{
    public static void main(String[] args)
    {
        PlayCircle circle = new PlayCircle();
        circle.setDiameter(2);
        System.out.println("If circle has diameter 2,");
        circle.showArea();

        System.out.println("Now you choose the diameter:");
        PlayCircle.areaDialog();
    }
}
```

屏幕对话示例

```
If circle has diameter 2,
Area is 3.14159
Now you choose the diameter:
Enter diameter of circle:
4
Area is 12.56636
```

图5-6 静态方法和非静态方法的使用

经常会听到人们说："不能在一个静态方法定义内部调用一个非静态方法。"但这种说法并不完全正确。更精确也更正确的说法是："除非为非静态方法创建并使用一个调用对象，否则不能在静态方法内部调用非静态方法。"换句话说，在静态方法的定义中，不能使用以隐式或显式的this作为调用对象的实例变量或方法。这是因为调用静态变量时可以不使用任何调用对象，因此它就可能会在this没有意义的情况下被调用。正如格特鲁德·斯泰因可能会说的那样，在静态方法中，"那里没有this。" △

■ Java提示：可以将main放到任意一个类中

到目前为止，每次在程序的main部分使用类时，main方法都位于另一个文件的不同类定义中。但有时在一个常规的类定义中包含main方法是有意义的。这样，就可以将这个类用于两个目的：可以在其他类中用它来创建对象，或者将它作为一个程序来运行。例如，可以将图5-6中显示的main方法放到PlayCircle的类定义中，通过这种方法把图5-5中的PlayCircle类定义和图5-6中的程序结合起来，获得图5-7所示的类定义。用这种方式重新定义了PlayCircle类之后，可以在另一个程序中将其作为一个普通类来创建PlayCircle类的对象，也可以将其作为一个程序来运行。将其作为一个程序运行时，会调用main方法，而除了main中用到的部分之外，类定义的其他部分都会被忽略。将其作为普通的类来创建对象时，会忽略main方法。

Java要求程序的main方法是静态的。因此，除非在main方法中创建了类对象，并用它作为非静态方法的调用对象，否则就不能在main方法内部调用同一个类中的非静态方法。注意图5-7中的main方法，它先创建了一个PlayCircle类的对象circle，并用circle作为调用对象调用了方法showArea和setDiameter。即使main方法位于PlayCircle类的定义内部，也必须这么做。

你可能不想在一个只会被用作普通类去创建对象的类定义中放置main方法。不过，有一种方便好用的小技巧，就是在类定义的main方法中放置一个小的诊断程序。

```
import java.util.*;

public class PlayCircle2
{
    public static final double PI = 3.14159;

    private double diameter;

    public static void main (String[] args)
    {
        PlayCircle circle = new PlayCircle();
        circle.setDiameter(2);
        System.out.println("If circle has diameter 2,");
        circle.showArea();
        System.out.println("Now you choose the diameter:");
        PlayCircle.areaDialog();
    }

    public void setDiameter(double newDiameter)
    {
```

在Web上可以找到的代码中，这个文件存放在PlayCircle2.java里。

由于这个main位于PlayCircle类定义的内部，所以如果想省略这个Playcircle，可以把它省略。

图5-7 在类定义中放置一个main方法

```
        diameter = newDiameter;
    }

    public static double area(double radius)
    {
        return (PI*radius*radius);
    }

    public void showArea()
    {
        System.out.println("Area is " + area(diameter/2));
    }

    public static void areaDialog()
    {
        Scanner keyboard = new Scanner(System.in);
        System.out.println("Enter the diameter of a circle:");
        double newDiameter = keyboard.nextDouble();
        PlayCircle c = new PlayCircle();
        c.setDiameter(newDiameter);
        c.showArea();
    }
}
```

图5-7 （续）

快速参考：静态方法

如果在方法定义的头部放置一个关键字static，就可以用类名取代调用对象来调用方法了。

由于静态方法不需要调用对象，所以它无法引用类的（非静态）实例变量，也无法调用类的非静态方法（除非它创建了一个新的类对象，并用此对象作为调用对象）。另一种表述方法是，在静态方法的定义中，不能使用将隐式或显式的this作为调用对象的实例变量或方法。

举例（静态方法的调用）：

```
int result = Math.max(n1, n2);
```

Math是一个预定义的类。

自测题

4. 下列代码是合法的吗（CircleFirstTry类是在图5-3中定义的）？

```
CircleFirstTry c = new CircleFirstTry();
double areaOfCircle = c.area(2.5);
```

5. 一个类中可以既包含静态方法又包含非静态方法（即常规方法）吗？它可以既包含实例变量，又包含静态方法吗？
6. 可以在一个静态方法内调用一个非静态方法吗？
7. 可以在一个非静态方法内调用一个静态方法吗？
8. 可以在一个静态方法内引用一个实例变量吗？为什么可以或者为什么不可以？
9. 下列代码是合法的吗？PlayCircle类是在图5-5中定义的。

```
Scanner keyboard = new Scanner(System.in);
```

```
System.out.println("Enter the diameter of a circle:");
double newDiameter = keyboard.nextDouble();
PlayCircle.setDiameter(newDiameter);
```

5.2.2 静态变量（选读）

类可以包含静态变量及静态方法。前面已经在一种特殊的情况下使用过静态变量了，例如：

```
public static final double PI = 3.14159;
```

这是在常量值的定义中使用静态变量。也有一些**静态变量**（static variable）的值是可以修改的。除了不使用单词final之外，这些静态变量的声明方式与已定义常量值的声明方式是一样的。因此，如果下列代码出现在类定义中，定义的就是一个值可以改变的静态变量：

```
public static double PI;
```

尽管这可能不是个好的静态变量名。

可以对静态变量进行初始化，也可以不对进行初始化；它可以是公有的也可以是私有的。但是，和实例变量一样，静态变量通常都是私有的，而且应该只能通过访问方法和设置方法进行读取和修改。因此，下面的代码是一个更恰当的静态（非常量的）变量声明示例：

```
private static int numberOfInvocations = 0;
```

变量numberOfInvocations被初始化为0。类中只有一个变量numberOfInvocations，类的每个对象都可以访问它。因此，这些对象可以通过这个静态变量来进行通信，或执行某些联合动作。例如，在图5-8的类定义中，用这个静态变量记录了StaticDemo类的所有对象总共进行了多少次调用。程序对图5-8中定义的、除了方法main之外的所有方法调用进行了计数。

```
public class StaticDemo
{
    private static int numberOfInvocations = 0;

    public static void main(String[] args)
    {
        int i;
        StaticDemo object1 = new StaticDemo();
        for (i = 1; i <=10 ; i++)
           object1.outPutCountOfInvocations();

        StaticDemo object2 = new StaticDemo();
        for (i = 1; i <=10 ; i++)
           object2.justADemoMethod();

        System.out.println("Total number of invocations = " + numberSoFar());
    }

    public void justADemoMethod()
    {
        numberOfInvocations++;
        //In a real example, more code would go here.
    }
```

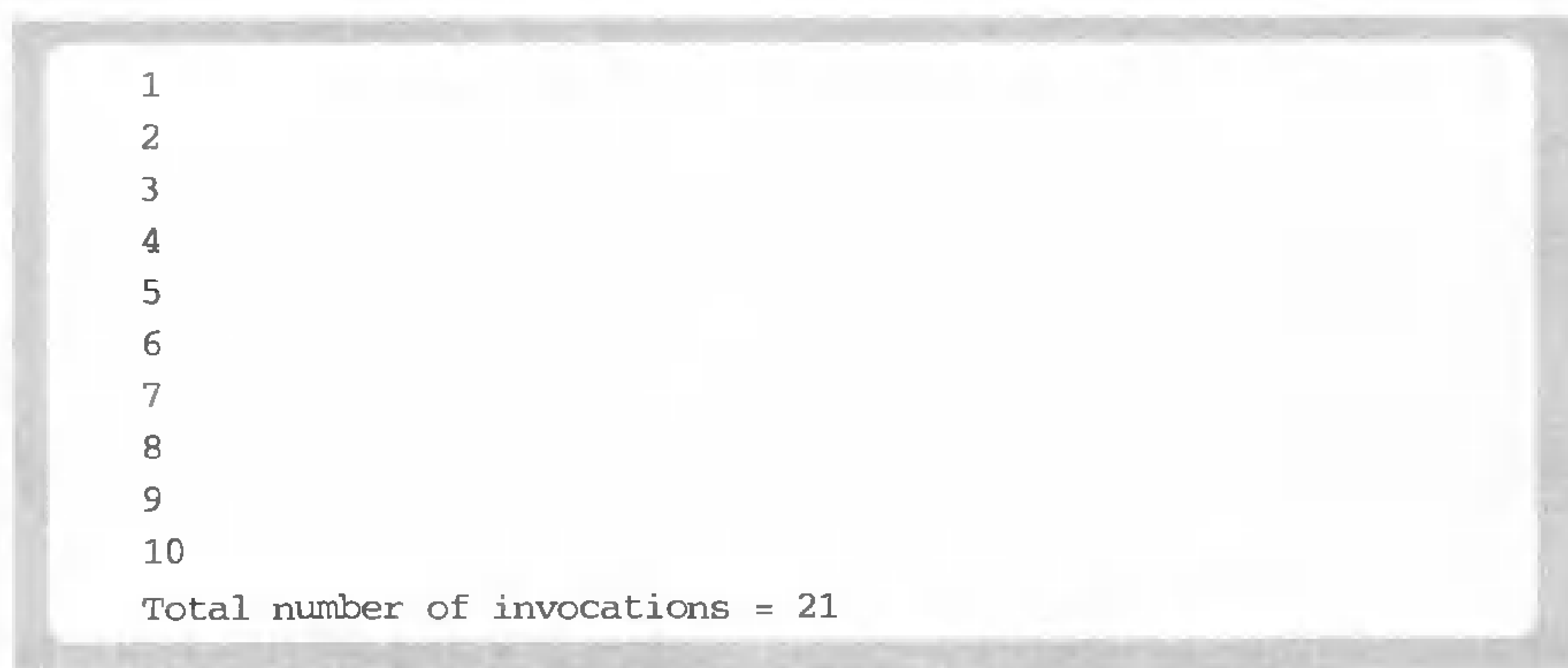
object1和object2使用了相同的静态变量numberOfInvocations。

图5-8 静态变量（选读）

```
    public void outputCountOfInvocations()
    {
        numberOfInvocations++;
        System.out.println(numberOfInvocations);
    }

    public static int numberSoFar()
    {
        numberOfInvocations++;
        return numberOfInvocations;
    }
}
```

屏幕对话示例



```
1
2
3
4
5
6
7
8
9
10
Total number of invocations = 21
```

图5-8 （续）

在有些情况下需要使用静态变量，但这种情况很少见，而且在结束了这里的讨论之后，不会再在本书中使用静态变量。

静态变量也被称为**类变量**（class variable）。不要将术语*类变量*（只是表示类中的一个静态变量）与类类型的变量（表示一个类型为类的变量，即用来命名类对象的变量）混淆。

自测题

10. 静态变量和实例变量之间有什么区别？
11. 可以在一个*静态方法*定义中（用名字，而不使用任何类名和点）引用*静态变量*吗？可以在一个*静态方法*定义中（用名字，而不使用任何对象名和点）引用*实例变量*吗？
12. 可以在一个*非静态方法*定义中（用名字，而不使用任何类名和点）引用*静态变量*吗？可以在一个*非静态方法*定义中（用名字，而不使用任何对象名和点）引用*实例变量*吗？

5.2.3 Math类

预定义的Math类提供了很多标准的数学方法。使用Java语言时自动提供这个类。图5-9对Math类中的一些方法进行了描述。所有这些方法都是静态的，这就意味着不需要（而且实际上也没必要）使用Math类的对象。通常通过类名Math代替调用对象来调用这些方法。例如，下列代码会输出2和3这两个数字中的最大值。

```
int ans;
ans = Math.max(2, 3);
System.out.println(ans);
```

省略变量ans，只写为下列形式也可以：

```
System.out.println("The maximum of 2 and 3 = " + Math.max(2,3));
```

名字	描述	实参类型	返回值类型	举例	返回值
pow	幂	double	double	Math.pow(2.0,3.0)	8.0
abs	绝对值	int、long、float或double	和实参的类型相同	Math.abs(-7) Math.abs(7) Math.abs(-3.5)	7 7 3.5
max	最大值	int、long、float或double	和实参的类型相同	Math.max(5,6) Math.max(5.5,5.3)	6 5.5
min	最小值	int、long、float或double	和实参的类型相同	Math.min(5,6) Math.min(5.5,5.3)	5 5.3
round	四舍五入	float或double	分别为int或long	Math.round(6.2) Math.round(6.8)	6 7
ceil	上取整	double	double	Math.ceil(3.2) Math.ceil(3.9)	4.0 4.0
floor	下取整	double	double	Math.floor(3.2) Math.floor(3.9)	3.0 3.0
sqrt	平方根	double	double	sqrt(4.0)	2.0

图5-9　Math类中的静态方法

Math类还有两个预定义常量E和PI。常量PI（在数学公式中通常写作π）用于包含圆、球体和其他基于圆的几何图形的计算中。PI近似为3.14159。常量E是自然对数的底数（在数学公式中通常写作e），近似为2.72。（在本书中没有用到预定义常量E。）常量PI和E是预定义常量。例如，下列代码在给定半径的情况下，计算出了圆的面积：

```
area = Math.PI * radius * radius;
```

注意，由于常量PI和E是在Math类中定义的，因此在它们前面一定要加上类名Math和一个点。例如，可以在图5-3所示的CircleFirstTry类定义中使用常量Math.PI。在图5-10中，用Math.PI重新编写了图5-3中的类定义。

在图5-9列出的方法中有3个类似但又不完全相同的方法：round、floor和ceil。有的方法会返回一个double类型的值，但它们返回的值看上去都是整数，而且与它们的实参值很接近。方法round会将一个数字四舍五入为最近的整数值，并且（如果实参为double类型）会将一个整数作为long类型的值返回。如果想将整数作为int类型的值返回，就要按下列形式使用强制类型转换[①]：

```
double start = 3.56;
int answer = (int)Math.round(start);
```

方法floor和ceil都与round类似，但又稍有不同。尽管这两种方法都会产生一个与其实参接近的整数，但它们都不是真正的四舍五入。这两个方法返回的都是double类型的（而

① 不能将long类型的值存储到int类型的变量中，即使是4这样和int类型完全一样的值也不行。例如，根据其创建方式的不同，值4可以是int类型的，也可以是long类型的。

不是int或long类型的）整数值。方法floor返回的是小于或等于其实参，且与其实参最接近的一个整数。因此Math.floor(3.9)返回的是3.0，而不是4.0。Math.floor(3.3)返回的也是3.0。

方法ceil返回的是大于或等于其实参，且与其实参最接近的整数。因此Math.ceil(3.1)返回的是4.0，而不是3.0。当然，Math.ceil(3.9)返回的也是4.0。

```
/**
 Class with static methods to perform calculations on circles.
*/
public class Circle
{
    public static double area(double radius)
    {
        return (Math.PI*radius*radius);
    }
    public static double circumference(double radius)
    {
        return (Math.PI*(radius + radius));
    }
}
```

Web上的源代码中的CircleDemo2.java是这个类的演示程序。

这个类的表现与图5-3中CircleFirstTry类的表现相同。这个版本唯一的不同在于，它使用了预定义常量Math.PI，而没有在类中定义PI。

图5-10 预定义常量

如果你想将floor或ceil返回的值存储在一个int类型的变量中，就要按下列形式使用强制类型转换：

```
double start = 3.56;
int lowAnswer = (int)Math.floor(start);
int highAnswer = (int)Math.ceil(start);
```

在这个例子中，Math.floor(start)返回的是double值3.0，变量lowAnswer收到的是int值3。Math.ceil(start)返回的是double值4.0，变量highAnswer收到的是int值4。

自测题

13. 下列代码都会返回什么样的值？

```
Math.round(2.3), Math.round(2.7), Math.floor(2.3),
Math.floor(2.7), Math.ceil(2.3), Math.ceil(2.7).
```

14. 假设speed是一个double类型的变量，要将Math.round(speed)赋给int类型的变量approxSpeed。赋值语句应该如何编写？

15. 假设speed是一个double类型的变量，要将Math.round(speed)赋给long类型的变量longSpeed。赋值语句应该如何编写？

16. 假设n1是int类型的变量，n2是long类型的变量。Math.min(n1,n2)的返回值会是什么类型？

5.2.4 Integer、Double及其他包装类

Java对基本类型（如int、double和char）和类类型（如String类和程序员定义的类）进行了区分。比如，第4章中根据实参是基本类型还是类类型，对方法实参的处理是不同的。为了使两者统一起来，有时要是能将基本类型的值，比如int值42，转换成某个对应于基本类型int的类类型的对象，会很方便。

为了将基本类型的值转换成“等价的”类类型值，Java为每个基本类型提供了**包装**（wrapper）类。例如，基本类型int的包装类为预定义类Integer。如果你想将一个int值，如42，转换成一个Integer类型的对象，可以用如下方式来实现：

```
Integer n = new Integer(42);
```

执行了这条语句之后，n就命名了一个对应于int值42的Integer类对象。（对象n实际上是将int值42存储在对象n的一个实例变量中了。）要进行反向转换，即从一个Integer类型的对象转换成一个int值，可以用如下方式来实现：

```
int i = n.intValue(); //n names an object of the class Integer.
```

方法intValue()会将等价的int值从Integer类型的对象中恢复出来。

基本类型long、float、double和char的包装类分别为Long、Float、Double和Character。当然，类Long、Float、Double和Character使用的不是方法intValue，而分别是方法longValue、floatValue、doubleValue和charValue。

正如“Java提示：自动装箱和拆箱”部分所述，在Java 5.0（或之后的版本）中，从int这样的基本类型向其相应的包装类（如Integer）的转换都是自动实现的。因此

```
Integer n = new Integer(42);
```

可以写成更简单的形式：

```
Integer n = 42;
```

但是，最后这条看起来很简单的赋值语句实际上就是对new Integer(42)这条较长语句的缩写。（这种情况与将一个int类型的值赋给一个double类型的变量时发生的情况类似；强制类型转换是自动完成的。

类似地，需要的时候，Java也会自动将包装类转换为其对应的基本类型的值。因此

```
int i = n.intValue(); //n names an object of the class Integer.
```

可以写成更简单的形式：

```
int i = n; // n names an object of the class Integer.
```

“Java提示：自动装箱和拆箱”部分对基本类型与其相应包装类之间相互进行的自动强制类型转换进行了介绍。

快速参考：包装类

每个基本类型都有一个包装类。包装类可以使你拥有一个对应于基本类型值的类对象。包装类中还包含了很多有用的预定义常量和静态方法。

这里对包装类的简要介绍说明了为什么要将其称为包装类，但对我们来说，最重要的是这些包装类包含了很多有用的常量和静态方法。

例如，可以用相关的包装类来查找任意基本数字类型中的最大值和最小值。int类型的最

大值和最小值为

```
Integer.MAX_VALUE 和 Integer.MIN_VALUE
```

double类型的最大值和最小值为

```
Double.MAX_VALUE 和 Double.MIN_VALUE
```

可以用包装类中的静态方法将一个字符串转换成相应的int、double、long或float类型的数字。例如，包装类Double的静态方法parseDouble会将一个字符串转换成一个double类型的值。因此

```
Double.parseDouble("199.98")
```

会返回一个double值199.98。当然，你知道那个数字值为199.98，看来好像不值得这么费力转换。但是，可以用同样的技术转换字符串变量的值。假设theString是一个String类型的变量，其值是一个用字符串表示的double类型的数字。下列代码就会返回对应于theString中字符串值的double值：

```
Double.parseDouble(theString)
```

如果theString命名的字符串中可能会有一些额外的前导及尾部空白，就应该使用下列语句：

```
Double.parseDouble(theString.trim());
```

方法trim是String类中的一个方法，可以消除前导及尾部空白，比如空格。

如果字符串中的数字形式不正确，对Double.parseDouble的调用就会使程序终止。使用trim可以在一定程度上避免发生这个问题。

这种从字符串到数字的转换可以用于任意一种包装类Integer、Long和Float，就像用于包装类Double一样。只是要用静态方法Integer.parseInt、Long.parseLong或者Float.parseFloat来取代Double.parseDouble。

每个数字包装类还有一个名为toString的静态方法，可以进行另一个方向的转换——将一个数字值转换成这个数字值的字符串表示。例如：

```
Integer.toString(42)
```

会返回字符串值"42"，而

```
Double.toString(199.98)
```

会返回字符串值"199.98"。

Character是基本类型char的包装类。下面这段代码说明了这个类的一些基本方法：

```
Character c1 = new Character('a');
Character c2 = new Character('A');
if (c1.equals(c2))
    System.out.println(c1.charValue() +
                       " is the same as " + c2.charValue());
else
    System.out.println(c1.charValue() +
                       " is not the same as " + c2.charValue());
```

输出内容如下：

```
a is not the same as A
```

equals方法会检查字符的相等性，因此大写和小写字母会被当作不一样的字符。

图5-11中列出了Character类中的部分静态方法。

名　字	描　述	实参类型	返回值类型	举　例	返回值
toUpperCase	转换成大写字符	char	Char	Character.toUpperCase('a') Character.toUpperCase('A')	都返回'A'
toLowerCase	转换成小写字符	char	Char	Character.toLowerCase('a') Character.toLowerCase('A')	都返回'a'
isUpperCase	检查是否是大写字符	char	boolean	Character.isUpperCase('A') Character.isUpperCase('a')	true false
isLowerCase	检查是否是小写字符	char	boolean	Character.isLowerCase('A') Character.isLowerCase('a')	false true
isWhitespace	检查是否是空白字符	char	boolean	Character.isWhitespace(' ') Character.isWhitespace('A')	true false
空白字符就是以空白形式打印的字符，如空格、tab字符（'\t'）以及换行符（'\n'）					
isLetter	检查是否是字母	char	boolean	Character.isLetter('A') Character.isLetter('%')	true false
isDigit	检查是否是数字	char	boolean	Character.isDigit('5') Character.isDigit('A')	true false

图5-11　Character类中的静态方法

还有一个名为Boolean的包装类。它有两个boolean类型的常量：Boolean.TRUE和Boolean.FALSE。但Java的关键字true和false更好用一些。

记住：包装类的两种特性

每个包装类都有两种相关但不同的用途。例如，对于包装类Integer，可以用它来生成与int类型值相对应的Integer类的对象：

```
Integer n = new Integer(42);
```

包装类Integer还可以作为一个包含了很多有用的静态方法的库来使用，比如下列代码中的parseInt方法：

```
String numeralString;
   ...
int number = Integer.parseInt(numeralString);
```

任何一个程序都可以使用包装类的这两种特性，但任意一个给定程序很可能只会使用其中的一种特性。

■ Java提示：自动装箱和拆箱

从基本类型值到与其相关的包装类对象之间的类型转换被称为**装箱**（boxing），如从int类型到Integer类型对象的转换。可以将对象当成一个包含了基本类型值（作为私有实例变量值）的“盒子”。本提示中后面所有的代码都是装箱的示例：

```
Integer n = new Integer(42);
Double d = new Double(9.99);
Character c = new Character('Z');
```

从5.0版开始，Java会自动地完成这种装箱过程，因此可以将前面3行赋值语句写成下列等效但更简单的形式：

```
Integer n = 42;
```

```
Double d = 9.99;
Character c = 'Z';
```

这里显示的是自动强制类型转换。Java实际完成的是带有new的代码段所示的工作。但是能用更简单的形式来书写赋值语句会非常方便。

从包装类对象到其相应的基本类型值的反向转换被称为**拆箱**（unboxing）。在Java 5.0之后，拆箱也是自动完成的。下列是一些自动拆箱示例：

```
int i = new Integer(42);
double f = new Double(9.99);
char s = new Character('Z');
```

实际发生的动作是Java自动地用适当的访问方法（在上述情况下分别为intValue、doubleValue或charValue）获得了赋给变量的基本类型值。

和简单赋值语句一样，自动装箱和拆箱也适用于各种参数。可以插入一个基本类型的值作为其相关包装类的参数，比如可以插入int类型的值作为Integer的参数。同样也可以插入一个包装类实参作为与其相关的基本类型参数，比如，可以插入Integer类型的实参作为一个int参数。

在讨论到Vector类之前，我们都没有机会使用装箱和拆箱。《Java程序设计语言与问题解决：高级篇》的第3章介绍Vector类时会重新对装箱和拆箱进行讨论。

自测题

17. 下列哪些代码是合法的？

```
Integer n = new Integer(77);
int m = 77;
n = m;
m = n;
```

如果有非法语句，说明如何编写一条合法的Java语句来完成那条非法语句想完成的任务。

18. 编写一个Java表达式，将double变量x中的数字转换成一个字符串。表达式返回一个用普通方式表示x值的字符串。
19. 编写一个Java表达式，将变量s中的字符串转换成相应的int类型的值。变量s是String类型的。假设s中包含了一个字符串，这个字符串用普通方式书写了一个整数，如"123"。
20. 如果字符串中可能包含前导或尾部空格，比如"123"，该如何完成自测题19？
21. 编写一段Java代码，输出在Java中能够得到的double类型的最大值和最小值。

5.3 设计的方法

遇到问题时，找一种方法来试试，这是常识。如果失败了，坦率地承认，然后再去试试另外一种方法。但首先，要去尝试。

——富兰克林·德拉诺·罗斯福，美国前总统，在奥格索普大学的演讲（1932年5月22日）

本节将介绍一些有助于设计并测试方法的基本技巧。下面从一个案例分析开始。

案例分析：对输出进行格式化

如果有一个存储了钱款数额的double类型变量，你可能希望程序能够以一种好的格式输出数额。但如果只使用System.out.println，很可能会得到下面的输出：

```
Your cost, including tax, is $19.98123576432
```

你可能希望输出看起来如下所示：

```
Your cost, including tax, is $19.98
```

在这个案例中，要定义一个名为Dollars的类，这个类带有两个名为write和writeln的静态方法，可以用这两个方法来生成这种格式良好的输出[①]。例如，如果钱款数额存放在一个名为amount的double类型变量中，那么，（完成了这个案例研究之后）可以用下列形式来生成输出：

```
System.out.print("Your cost, including tax, is ");
Dollars.writeln(amount);
```

注意，这个方法会添加美元符号，而且在小数点后面总是会精确地输出两个数字。因此，它会输出$2.10，而不是$2.1。

当要写出数额的小数点后面的数字多于两位时，你必须决定如何处理多余的数字。将小数点后面的数字四舍五入为两位，那么如果要输出的值为9.128，得到的输出是$9.13。

write和writeln之间的区别与print和println之间的区别是一样的。使用write时，接下来的输出会出现在同一行；使用writeln时，接下来的输出会出现在下一行。

要输出一个9.98这样的数额，除了将它分成9和98两个部分，然后将每个部分单独输出之外，没有其他选择。因此，提出了下列伪代码作为write方法的轮廓：

将一个double类型的数额以美元和美分的形式输出的算法

（数额存储在一个名为amount的变量中）

确定amount中完整的美元数，并将其存储在一个名为dollars的int变量中。

确定amount中的美分数，并将其存储在一个名为cents的int变量中。

如果小数点后的数字多于两位，就对其进行四舍五入。

```
System.out.print('$');
System.out.print(dollars);
System.out.print('.');
```

以通常的美元和美分格式输出cents。

现在要将这里的每一条伪代码指令都转换成Java代码，按顺序来对其进行转换。

要想得到表示完整的美元数和美分数的两个int值，就需要通过某种方式去除小数点。去除小数点的一种方法是把钱款额全部转换成美分。要把用美元和美分表示的10.95转换成全部用美分表示的形式，就要将其乘以100，得到10.95*100，即1095.0。如果转换之后美分中还存在分数，比如把10.9567转换成1095.67（美分）就存在分数，这时候可以使用round方法：

```
Math.round(1095.67);
```

这条语句会把1096作为一个long类型的值返回。

注意，要把这次四舍五入的结果存储在一个int变量中，这个int变量是用来包含全部由美分表示的总额的。但是，Math.round返回的是一个long类型的值，但不能将long类型的值存储在int类型的变量中，即使是个很小的整数也不行。因此，要按下列方法执行一次强制类型转换，将long值转换成int值：

```
(int)(Math.round(1095.67))
```

这条语句会把1096作为一个int值返回。

因此，代码应该以

① Java中有一些类允许你按照自己的意愿用任意的格式来输出数字。但这些类的使用很复杂。而自己来编写输出的细节会更有意义，可能也更容易一些。如果想了解更多与这些格式化类有关的内容，可以查看附录F，其中简要介绍了DecimalFormat类。

```
int allCents = (int)(Math.round(amount*100));
```

这样的语句开始。

现在要将allCents转换成一个美元数和一个美分数。一美元包含100美分，因此可以通过整数除法来获得美元数：

```
int dollars = allCents/100;
```

美分数就是allCents被100除之后剩余的数额，因此，可以用%运算符来获得美分数：

```
int cents = allCents%100;
```

这样，就可以将伪代码中的前两步翻译成下列Java代码了：

```
int allCents = (int)(Math.round(amount*100));
int dollars = allCents/100;
int cents = allCents%100;
```

伪代码中接下来的3行已经是用Java表示的了。因此，剩下的工作就是将最后一条指令转换成Java代码。最后一条指令为：

以通常的美元和美分格式输出cents。

这看起来很容易。采用下列语句：

```
System.out.println(cents);
```

然后测试你的代码，会得到下列输出：

```
$10.96
```

这看起来很不错。但当尝试其他一些情况时，就会在数额小于10时出现问题。例如，在期望得到$7.05时，得到的输出为

```
$7.5
```

通过这种快速测试会发现，当美分数小于10时，需要在美分数的前面输出一个0。因此，要将输出美分数的代码修改为：

```
if (cents < 10)
{
    System.out.print('0');
    System.out.print(cents);
}
else
    System.out.print(cents);
```

图5-12显示了你导出的完整类定义。

```
public class DollarsFirstTry
{
    /**
     Outputs amount in dollars and cents notation.
     Rounds after two decimal points.
     Does not advance to the next line after output.
    */
    public static void write(double amount)
    {
        int allCents = (int)(Math.round(amount*100));
        int dollars = allCents/100;
        int cents = allCents%100;
        System.out.print('$');
```

图5-12 DollarsFirstTry类

```
        System.out.print(dollars);
        System.out.print('.');
        if (cents < 10)
        {
            System.out.print('0');
            System.out.print(cents);
        }
        else
            System.out.print(cents);
    }
    /**
     Outputs amount in dollars and cents notation.
     Rounds after two decimal points.
     Advances to the next line after output.
    */
    public static void writeln(double amount)
    {
        write(amount);
        System.out.println();
    }
}
```

图5-12 （续）

现在既然有了完整的类定义，就该进行一些认真的测试了。图5-13显示的是用来测试方法write的程序。因为这类程序除了测试（“驱动”）方法之外，什么事都不做，所以通常将它们称为**驱动程序**（driver program）。任何方法都可以在这样的程序中进行测试。

```
import java.util.*;

public class DollarsFirstTryDriver
{
    public static void main(String[] args)
    {
        double amount;
        String ans;
        Scanner keyboard = new Scanner(System.in);

        System.out.println("Testing DollarsFirstTry.write:");
        do
        {
            System.out.println("Enter a value of type double:");
            amount = keyboard.nextDouble();
            DollarsFirstTry.write(amount);
            System.out.println();
            System.out.println("Test again?");
            ans = keyboard.next();
        }while (ans.equalsIgnoreCase("yes"));
        System.out.println("End of test.");
    }
}
```

这类测试程序通常被称为驱动程序。

图5-13 测试方法

屏幕对话示例

```
Testing DollarsFirstTry, write:
Enter a value of type double:
1.2345
$1.23
Test again?
yes
Enter a value of type double:
1.235
$1.24
Test again?
yes
Enter a value of type double:
9.02
$9.02
Test again?
yes
Enter a value of type double:
-1.20
$-1.0-20
Test again?
no
```

这儿有点儿问题。

图5-13 （续）

测试一直进行的很好，直到你决定试着使用一个负数为止。毕竟，会有钱数为负的情况出现。这种情况被称为负债。但钱数-1.20会被输出成$-1.0-20。你的方法处理负数的方式存在问题。

很容易发现出了什么问题。输出$-1.之后，你想输出的是20，而不是-20。有很多方法可以修正这个问题，但要找一种清晰且简洁的方式。因为已经有了可以正确输出非负数的代码，所以你可以将任意一个负数转换成正数，然后输出正数，再插入一个负号。图5-14显示了这个类的修正版本。注意，其新方法writePositive的主体与图5-12中write方法的主体基本一样。唯一的区别是writePositive没有输出美元符号。美元符号是在新版的write方法中输出的。

```
public class Dollars
{
    /**
     Outputs amount in dollars and cents notation.
     Rounds after two decimal points.
     Does not advance to the next line after output.
    */
    public static void write(double amount)
    {
        if (amount >= 0)
        {
            System.out.print('$');
            writePositive(amount);
        }
        else
```

图5-14 经过校正的Dollars类

```
        {
            double positiveAmount = -amount;
            System.out.print('$');
            System.out.print('-');
            writePositive(positiveAmount);
        }
    }

    //Precondition: amount >= 0;
    //Outputs amount in dollars and cents notation. Rounds
    //after two decimal points. Omits the dollar sign.
    private static void writePositive(double amount)
    {
        int allCents = (int)(Math.round(amount*100));
        int dollars = allCents/100;
        int cents = allCents%100;
        System.out.print(dollars);
        System.out.print('.');
        if (cents < 10)
        {
            System.out.print('0');
            System.out.print(cents);
        }
        else
            System.out.print(cents);
    }

    /**
     Outputs amount in dollars and cents notation.
     Rounds after two decimal points.
     Advances to the next line after output.
    */
    public static void writeln(double amount)
    {
        write(amount);
        System.out.println();
    }
}
```

钱数为负的情况。

Web上源代码中的DollarsDriver.java是这个类的测试及演示程序。

图5-14 （续）

每次修改类或方法定义时，都应该对其进行测试。因此，要用一个程序来重新测试Dollars类，这个程序与测试DollarsFirstTry的程序类似。这次，测试是成功的。

Dollars类中write方法的驱动程序位于文件DollarsDriver.java中，Web上本书的源代码中包含了这个文件。

5.3.1 自顶向下的设计

在上面的案例分析中，使用了下列伪代码作为设计图5-14中Dollars类中write方法的第一次尝试：

确定amount中完整的美元数，并将其存储在一个名为dollars的int变量中。

```
确定amount中的美分数，并将其存储在一个名为cents的int变量中。
如果小数点后的数字多于两位，就对其进行四舍五入。
System.out.print('$');
System.out.print(dollars);
System.out.print('.');
以通常的美元和美分格式输出cents。
```

通过这段伪代码将输出一定数额钱款的任务分解成了很多的子任务，比如：

```
确定amount中完整的美元数，并将其存储在一个名为dollars的int变量中。
```

然后独立地解决每个子任务，并为每个子任务产生代码。毕竟，为了产生方法的最终定义，要完成的所有工作就是将这些子任务的代码结合起来。

最后的结果是，将从前面的伪代码中导出的代码用在了方法writePositive，而不是方法write中。但这正是应用子任务的一种更好的说明。方法writePositive解决了在方法write的最终定义中用到的一个子任务。

虽然不总是这样，但子任务通常都是作为私有辅助方法来实现的。如果子任务很大，就可以使用相同的技术。将子任务分成更小的子子任务，并单独地解决这些子子任务。这些子子任务还可以进一步分解为更小的任务，但最终这些任务会变得足够小，使其设计和代码实现都变得很容易。

这种将一个方法所要执行的任务划分成若干子任务的技术被称为**自顶向下的设计**（top-down design）或**分治**（divide and conquer）。

5.3.2 对方法的测试

对方法进行测试的一种方式是使用图5-13中所示的驱动程序。这些驱动程序只是给你自己用的，所以可以写得很简单，不需要包含任何花哨的输出或其他东西。这些程序所要做的就是给方法一些实参，来调用方法。

为类编写的每个方法都要进行测试。而且，在要测试的程序中，这个方法应该是唯一一个还没有经过完全测试的方法。这样，如果发现程序出了问题，就可以知道是哪个方法存在问题。如果在同一个程序中对多个方法进行测试，就很容易会在一个方法出现问题时，错误地认为问题出在另一个方法上。

记住：单独对方法进行测试

测试每个方法时，在测试程序中，这个方法都应该是唯一的未测试方法。

自底向上测试（bottom-up testing）是一种将方法放在一个它是唯一未测方法的程序中进行测试的技术。采用自底向上的测试方法时，如果方法A使用了方法B，那么，在测试方法A之前一定要对方法B进行完整的测试。

自底向上的测试是一种很好并且很安全的测试方法，但有时这种方法会很冗长乏味。使用其他一些方法可以更快更容易地找出问题。有时要在一个方法中用到的所有其他方法都测试完毕之前，对这个方法进行测试。但仍然应该在一个它是唯一未测方法的程序中测试这个方法。比如，在编写出所有方法之前，你可能就想对解决问题的基本方式进行测试了。有时会发现自己处于这样一种境地：方法A使用了方法B，但想在测试方法B之前（甚至可能是在

编写方法B之前）测试方法A。这就存在一个问题：如果方法A使用了方法B，而方法B还没有测试过，那么怎样才能在方法A是唯一未测方法的程序中对其进行测试呢？答案是为方法B使用一个**存根**（stub）。

存根是方法的一种简化，对最终的类定义来说它不够好，但对测试来说是够用的，而且它很简单，你可以确定（或者尽可能地确定）它是正确的。例如，假设你在测试图5-14中的Dollars类，而且想在测试方法write之前就测试方法writeln，可以为方法write使用下列存根：

```
public static void write(double amount)
{
     System.out.print("$99.12");
}
```

显然，这并不是一个正确的write类定义。无论收到什么样的实参，总会输出$99.12。但对方法writeln的测试来说，这个方法已经足够好。如果用write方法的这个存根来测试方法writeln，而且writeln以正确的方式输出了99.12，就基本上可以确定writeln是正确的。注意，write的这个存根的使用甚至可以让你在编写方法write或writePositive之前测试方法writeln。

自测题

22. 设计一个类来输出double类型的值（不一定是钱款数的输出）。调用OutputFormat类，这个类应该有两个静态方法：write和writeln，每个方法都有两个实参。第一个实参给出了一个要写到屏幕上的double值。第二个实参是一个int值，用来说明要在小数点后面显示几位数字。方法要将任何多余的数字四舍五入。这个方法与Dollars类中的write方法和writeln方法非常相似，你可以将Dollars类作为模板，但这个类的两个方法之间还是有些区别：write之后的输出会出现在同一行，而writeln之后的输出会出现在下一行；否则，write和writeln就是一样的了。下面是一些输出示例：

```
OutputFormat.writeln(9.1234667, 4);
OutputFormat.writeln(9.9999, 2);
OutputFormat.writeln(7.01234, 4);
```

上述代码会产生下列输出：

```
9.1235
10.00
7.0123
```

不要忘了用1.023和1.0023这样小数点后面有零的数字来测试你的方法。（提示：在删除小数点的表达式中使用图5-9中的静态方法Math.pow会有所帮助）。这是个相当难的练习，给自己一些时间来完成它。如果没有成功地编写出这个类，至少要确定你理解了本章结尾给出的答案。这是个很有用的类。

23. 在自测题22给出的OutputFormat类定义中，可以使用print和println，而不是write和writeln吗？这样会和System.out.prinln产生名字冲突么？

24. 图5-14方法writePositive中的变量allCents包含了全部由美分表示的钱款数。因此，对数额$12.95来说，应该将int值allCents设置为1295。这种方式有些局限性。int类型的最大值为2 147 483 647，这就意味着方法可以处理的最大数额为$21,474,836.47。这个数超过2100万美元，是很大一笔钱，但我们通常需要考虑更大的数额，比如国家的预算，一个大公司的预算，甚至

一个大公司CEO的薪水。如何才能通过对Dollars类中的方法定义的简单修改，使其能够处理更大的钱款数额呢？

5.4 重载

美名胜过贵重的膏油…… ——传道书7：1

两个（或多个）不同的类可以拥有同名的方法。例如，很多类中都会有一个名为readInput的方法。调用对象的类型允许Java来确定应该使用哪个readInput方法定义。实际上，可以在同一个类中拥有两个或多个具有相同名字的方法。

重载的基本概念

在同一个类中给出两个或多个具有相同方法名的定义，称为方法名的**重载**（overload）。为了使其正常工作，必须确保这个方法名的不同定义中的参数列表会有所不同。例如，图5-15中包含了一个非常简单的重载示例。

```
/**
 This is just a toy class to illustrate overloading.
*/
public class Statistician
{
    public static void main(String[] args)
    {
        double average1 = Statistician.average(40.0, 50.0);
        double average2 = Statistician.average(1.0, 2.0, 3.0);
        char average3 = Statistician.average('a', 'c');
        System.out.println("average1 = " + average1);
        System.out.println("average2 = " + average2);
        System.out.println("average3 = " + average3);
    }

    public static double average(double first, double second)
    {
        return ((first + second)/2.0);
    }

    public static double average(double first, double second, double third)
    {
        return ((first + second + third)/3.0);
    }

    public static char average(char first, char second)
    {
        return (char)(((int)first + (int)second)/2);
    }
}
```

屏幕对话示例

```
average1 = 45.0
average2 = 2.0
average3 = b
```

图5-15 重载

图5-15中所示的Statistician类中有3个不同的方法，都叫作average。调用Statistician.average时，Java怎样才能识别出应该使用哪个average定义呢？首先，我们假设实参都是double类型的。在这种情况下，Java可以根据实参的数量来判定应该使用哪个average定义。如果有两个double类型的实参，它就会使用第一个average定义。如果有3个实参，会使用第二个average定义。

现在，假设一个average方法调用有两个char类型的实参。根据实参的类型，Java知道应该使用average的第3个定义。只有一种定义具有两个char类型的实参。现在，不考虑这个方法如何计算两个字母的平均值，以后再讨论。

假设你通过为同一个类定义中的一个方法名编写多个定义，重载了这个方法名。那么，在调用这个方法名时，Java会根据实参的数量和类型来确定使用哪个定义。如果一个方法名定义的参数数量与调用中实参的数量相同，而且类型也都相符，即如果第一个实参的类型与第一个参数的类型相同，第二个实参的类型与第二个参数的类型相同，以此类推，那么，这个定义就是Java要使用的方法名定义。如果没有相匹配的方法名定义，Java会尝试进行一些简单的强制类型转换，比如将一个int类型值转换成double类型值，看这样能否产生匹配的定义。如果这样还是不行，就会给出一条错误消息。

现在，简单解释一下如何计算两个字符的平均值。对这个例子来说，我们是否用了一种疯狂的方式来计算两个字符的平均值是无关紧要的，但实际上，我们在图5-15中所使用的技术是计算两个字符平均值的一种很合理的方式，至少是计算两个字母平均值的一种很合理的方式。如果两个字母都是小写的，计算出来的平均值就是按字母表排序时，这两个字母正中间的小写字母。（如果正中间没有字母，它会从最靠近中间位置的两个字母中选择一个。）类似地，如果两个字母都是大写字母，计算出来的平均值就是按字母表排序时两个字母正中间的大写字母。因为分配给字母的数字都是按顺序排列的，所以这种方法是可行的。分配给'b'的数字比分配给'a'的数字大1，分配给'c'的数字比分配给'b'的数字大1，以此类推。因此，如果将两个字母转换成数字，计算数字的平均值，再将得到的数字转换成字母，就能得到两个字母中间的字母了。

重载可以应用于任何类型的方法，可以用于void方法、返回值的方法、静态方法、非静态方法或者这些方法的任意组合。

注意，虽然以前可能不知道这个术语，但你已经在使用重载这个概念了。在5.3节中，Math类的很多方法都使用了重载。例如，max方法使用了基于其实参类型的重载。如果它的两个实参都是int类型的，就会返回一个int类型的值；如果两个实参都是double类型的，就会返回一个double类型的值。当然，这并不是重载最重要的应用，因为除了一些类型名之外，不同的max定义是完全一样的。一个更引人注目的例子是在第2章讨论过的除法运算符/。如果它的实参是double类型的，它就被定义为执行浮点除法，这样5.0/2.0就会返回2.5；但如果两个实参都是int类型的，它就被定义为执行整数除法，这样5/2就等于2。

快速参考：重载

在一个类中，一个方法名可以有两个（或多个）定义，称为**重载**方法名。对一个方法名进行重载时，同一个方法名的任意两个定义的参数个数都必须不同，或者两个定义中的某个位置上的参数必须是不同类型的。

编程示例：Pet类

图5-16显示了一个名为Pet的类的类图。注意，它有4个名为set的方法。这是一个重载方法名的示例。用各种不同的set方法来设置不同的实例变量。其中，有一个方法对所有的3个实例变量name、age和weight都进行了设置。其他3个方法都只设置了一个实例变量。

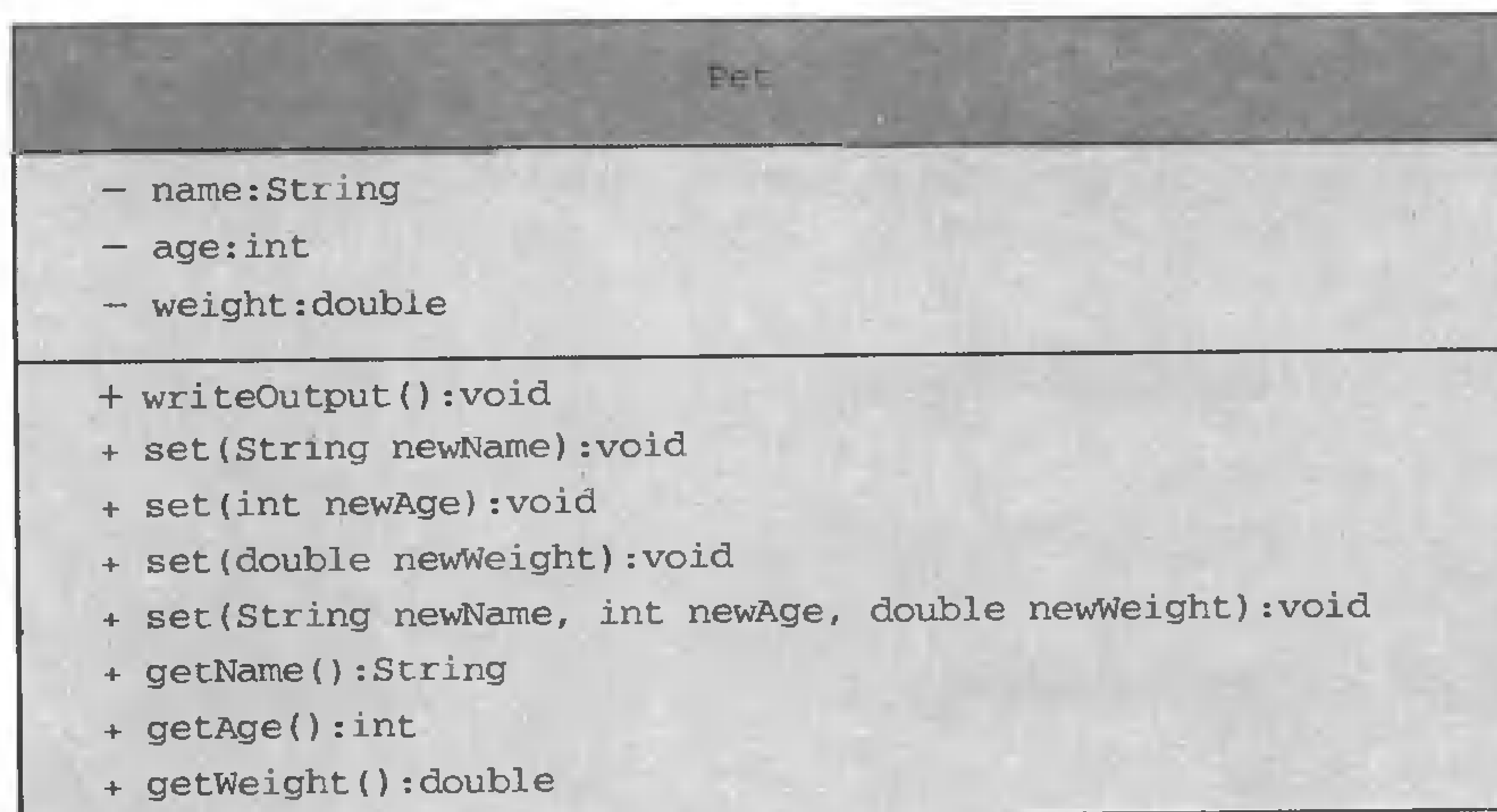


图5-16 Pet类的类图

图5-17显示了Pet类的完整定义。其中有一个带有简单演示程序的main方法。注意，每次进行set方法调用，或者实参数有所不同，或者有一个实参的类型与其他set调用不同。因此，每次进行set调用都使用了不同的set定义。

```
/**
 Class for basic pet records: name, age, and weight.
*/
public class Pet
{
    private String name;
    private int age; //in years
    private double weight; //in pounds

    /**
     This main is just a demonstration program.
    */
    public static void main(String[] args)
    {
        Pet myDog = new Pet();
        myDog.set("Fido", 2, 5.5);
        myDog.writeOutput();
        System.out.println("Changing name.");
        myDog.set("Rex");
        myDog.writeOutput();
        System.out.println("Changing weight.");
        myDog.set(6.5);
```

图5-17 Pet类

```
        myDog.writeOutput();
        System.out.println("Changing age.");
        myDog.set(3);
        myDog.writeOutput();
    }

    public void writeOutput()
    {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age + " years");
        System.out.println("Weight: " + weight + " pounds");
    }

    public void set(String newName)
    {
        name = newName;
        //age and weight are unchanged.
    }

    public void set(int newAge)
    {
        if (newAge <= 0)
        {
            System.out.println("Error: invalid age.");
            System.exit(0);
        }
        else
            age = newAge;
        //name and weight are unchanged.
    }

    public void set(double newWeight)
    {
        if (newWeight <= 0)
        {
            System.out.println("Error: invalid weight.");
            System.exit(0);
        }
        else
            weight = newWeight;
        //name and age are unchanged.
    }

    public void set(String newName, int newAge, double newWeight)
    {
        name = newName;
        if ((newAge <= 0) || (newWeight <= 0))
        {
            System.out.println("Error: invalid age or weight.");
            System.exit(0);
        }
```

图5-17　(续)

```
        else
        {
            age = newAge;
            weight = newWeight;
        }
    }

    public String getName()
    {
        return name;
    }

    public int getAge()
    {
        return age;
    }

    public double getWeight()
    {
        return weight;
    }
}
```

屏幕对话示例

```
Name: Fido
Age: 2 years
Weight: 5.5 pounds
Changing name.
Name: Rex
Age: 2 years
Weight: 5.5 pounds
Changi ng weight.
Name: Rex
Age: 2 years
Weight: 6.5 pounds
Changing age.
Name: Rex
Age: 3 years
Weight: 6.5 pounds
```

图5-17　（续）

▲ 易犯错误：重载与自动类型转换

在某些情况下，有两个朋友可能还不如有一个朋友好。两件好事有时会以一种不好的方式相互影响。重载是一个朋友，或者至少是Java语言中一种很有用的特性。实参的自动类型转换也是Java语言中一种很有用的特性（比如，在方法需要double类型的实参时，将2这样的int值转换成2.0这样的double值）。但这两种很好的特性有时候却会互相干扰。

例如，看看图5-17的main方法，并考虑下列代码：

```
System.out.println("Changing weight.");
```

```
myDog.set(6.5);
```

这段代码会将myDog的重量改成6.5磅。但是，假设myDog的重量不是6.5磅，而只有6磅。在这种情况下，应该将这两行改成：

```
System.out.println("Changing weight.");
myDog.set(6.0);
```

这样就可以将myDog的重量改成6磅了。但假如忘了写小数点和零，写下了下列代码：

```
System.out.println("Changing weight.");
myDog.set(6);
```

希望这段代码将myDog的重量改成6，但它却会将myDog的年龄改成了6。这是因为6是int类型的，而具有一个int类型参数的set定义会修改实例变量age，而不是修改实例变量weight。如果Java能找到一个与实参的数量和类型都相符的set定义，就不会进行从int到double，或任意其他类型之间的类型转换了。

在上述情况下，需要进行一次类型转换而没有进行。还有一些情况是你不希望进行类型转换，它却进行了。例如，假设想将myDog的名字改成"Cha Cha"，重量改成2，年龄改成3。我们可能会尝试下列语句：

```
myDog.set("Cha Cha", 2, 3);
```

这条语句会将myDog的年龄改成2，而不是3，将myDog的重量改成3.0，而不是2.0。当然，实际的问题在于颠倒了第二个和第三个实参，但是，看看Java接下来干了些什么。给出了前面的调用之后，Java会查找具有下列格式头部的set定义：

public void set(String *Name_1*, int *Name_2*, int *Name_3*)

但是，并没有这样的set定义。因此就没有与调用完全相符的set定义。然后，Java会试着将int类型值转换成double类型值，以获得一个相符的定义。它注意到如果将3转换成3.0，就会与下列语句匹配：

```
public void set(String newName, int newAge, double newWeight)
```

因此，进行了类型转换。

除了颠倒了两个实参的位置之外，还有什么地方出问题了呢？应该给出2.0而不是2作为重量。如果我们用的是2.0，或者Java没有进行任何自动类型转换，我们就会收到一条错误消息。在这种情况下，Java想帮忙，但却帮了倒忙。

在有些情况下，根据重载和类型转换之间的交互情况，方法调用可能会以两种不同的方式进行。Java中不允许出现这种有歧义的方法调用，而且会产生一个编译器错误消息。例如，可以对一个方法名（如problemMethod）进行重载，使其在SampleClass中具有下列两种方法头部：

```
public class SampleClass
{
    public static void problemMethod(double n1, int n2)
        .
        .
        .
    public static void problemMethod(int n1, double n2)
        .
        .
        .
```

这样的方法定义是可编译的。但是下面这样的方法调用就会产生一条错误消息，因为Java无法确定应该使用哪个problemMethod重载定义：

```
SampleClass.problemMethod(5, 10);
```

Java无法确定它是应该将int值5转换成一个double值以使用第一个problemMethod定义，还是应该将int值10转换成double值以使用第二个定义。在这种情况下，Java会产生一条错误消息，说明这个方法调用是有歧义的。

下面两个方法调用是可用的：

```
SampleClass.problemMethod(5.0, 10);
SampleClass.problemMethod(5, 10.0);
```

尽管这些调用是可用的，但很容易发生混淆，所以应该尽量避免。 △

记住：重载与自动类型转换

Java总是会在尝试进行自动类型转换之前尝试进行重载。如果Java能找到一个与实参类型相匹配的方法定义，就会使用那个定义。只有在Java试图寻找一个参数类型与方法调用中的实参完全匹配的方法名定义失败之后，才会对方法实参进行自动类型转换。

▲ 易犯错误：不能在返回值的基础上重载

不能通过给出两个仅在头部返回值类型有所区别的方法定义来对方法名进行重载。例如，对于图5-17中的Pet类，可能希望添加一个名为getWeight的方法，来返回一个说明宠物是超重还是重量不足的字符，比如，用'+'表示超重，用'-'表示重量不足，用'*'表示正好。这个方法会返回一个char类型的值。如果你添加了这个getWeight方法，就会有两个具有下列头部的方法：

```
/**
Returns the weight of the pet.
*/
public double getWeight()
/**
  Returns '+' if overweight, '-' if
  underweight and '*' if weight OK.
*/
public char getWeight()
```

不能在一个类中同时包含这两个方法。

遗憾的是，这是非法的。在任何类定义中，同一个方法名的两个定义都必须具有不同数量的参数或者有一个或多个类型不同的参数。不能在返回值的基础上进行重载。

要想编写出一个能在返回值的基础上进行重载的编译器都是不可能的。例如，有下列代码：

```
Pet myFriend = new Pet();
        .
        .
        .
double value = myFriend.getWeight();
```

现在，假设与实际情况相反，我们允许具有上述头部的两个方法存在。考虑一下在这种情况下编译器要完成的工作。尽管还没讨论过这个问题，但你确实可以将一个char类型的值存储在一个double类型的变量中。Java会执行自动强制类型转换将char类型值转换成double类型值。因此，在这种假想情况下，变量value可以接受一个double类型值，也可以接受一个char类型值。因此，就无法分辨出编写这些代码的程序员是希望getWeight返回一个char类型值，还是一个double类型值。编译器得问问程序员到底是什么意思，但是是不允许编译器向程序员提问的。 △

自测题

25. 可以在图5-17的程序中包含下列方法调用吗?

```
myDog.set("Fido", 2, 7);
```

26. 一个类中可以包含下面这两个方法定义吗?

```
/**
 Postcondition: Returns the number of people
 in numberOfCouples couples.
*/
public static int howMany(int numberOfCouples)
{
     return 2*numberOfCouples;
}
/**
 Postcondition: Returns the number of children,
 assuming that each couple has 2.3 children.
*/
public static double howMany(int numberOfCouples)
{
     return 2.3*numberOfCouples;
}
```

27. 一个类中可以包含下面这两个方法定义吗?

```
/**
 Postcondition: Returns an int value approximately
 equivalent to number.
 But converts all negative numbers to zero.
*/
public static int convertedValue(double number)
{
     if (number > 0.0)
          return (int)number;
     return 0;
}
/**
 Postcondition: Returns a double value approximately
 equivalent to number.
 But converts all negative numbers to zero.
*/
public static double convertedValue(int number)
{
     if (number > 0)
          return (double)number;
     return 0.0;
}
```

28. 图4-19中的Species类有一个名为set的方法，用来设置物种的名字、种群规模和增长率。假设有另一个也叫set的方法，这个方法只有一个用作物种名称的参数，并且将种群规模和增长率都设置为零，这个类中可以包含一个这样的方法吗？如果可以，给出这个set方法的定义。

29. 图4-19中的Species类有一个名为set的方法，用来设置物种的名字、种群规模和增长率。假设有另一个也叫set的方法，这个方法只有一个用作物种名称的参数，并且不对任何其他实例变量进行设置，这个类中可以包含一个这样的方法吗？如果可以，给出这个set方法的定义。

30. 对于图4-19中的Species类，可以将自测题28和29中定义的两个名为set的方法都添加到Species类中去吗?

编程示例：Money类

图5-18中包含了一个名为Money的类，这个类的对象表示了（美元）钱款数额，如$9.99、$500.00、$0.50等。你可能倾向于认为钱款数额是一个double类型的值，但不了解编程概念的用户（很多终端用户都不了解编程的概念）会把这些数额当作美元和美分。对“马路上那些人”来说，$9.99不是一个double类型的值。结果证明“马路上那些人”是对的。有时可以侥幸地用double类型的值来表示钱款数额，但这样做的时候会存在一个问题。实际上，double类型的值是一个近似的量值，如果编写的是一个记账程序，使用近似量值是不够好的。如果银行对一个账户余额的计算出现了几美元，甚至几美分的偏差，都会使客户不满意，并可能会引发客户或政府的一些法律行为。Money类是用来包含表示钱款的数据的。使用Money类的程序员，或者产生任何软件产品的终端用户，都不会把数据当成double、int或任何其他Java预定义类型的值，它们会被当成Money类型的值[①]。

```
import java.util.*;

/**
 Objects represent nonnegative amounts of money,
 such as $100, $41.99, $0.05.
*/
public class Money
{
    private long dollars;
    private long cents;

    public void set(long newDollars)
    {
        if (newDollars < 0)
        {
            System.out.println(
                 "Error: Negative amounts of money are not allowed.");
            System.exit(0);
        }
        else
        {
            dollars = newDollars;
            cents = 0;
        }
    }

    public void set(double amount)
    {
        if (amount < 0)
        {
            System.out.println(
                 "Error: Negative amounts of money are not allowed.");
            System.exit(0);
        }
```

图5-18 Money类

① 在图5-14中，定义了一个名为Dollars的类，这个类是将钱款数额作为double类型的值处理的。这个编程示例显示了另一种处理钱款数额的方法，与Dollars类没有直接的关系。

```
        else
        {
            long allCents = Math.round(amount*100);
            dollars = allCents/100;
            cents = allCents%100;
        }
    }

    public void set(Money otherObject)
    {
        this.dollars = otherObject.dollars;
        this.cents = otherObject.cents;
    }

    /**
     Precondition: The argument is an ordinary representation
     of an amount of money, with or without a dollar sign.
     Fractions of a cent are not allowed.
    */
    public void set(String amountString)
    {
        String dollarsString;
        String centsString;
        //Delete '$' if any:
        if (amountString.charAt(0) == '$')
            amountString = amountString.substring(1);
        amountString = amountString.trim();

        //Locate decimal point:
        int pointLocation = amountString.indexOf(".");

        if (pointLocation < 0) //If no decimal point
        {
            cents = 0;
            dollars = Long.parseLong(amountString);
        }
        else //String has a decimal point.
        {
            dollarsString =
                    amountString.substring(0, pointLocation);
            centsString =
                    amountString.substring(pointLocation + 1);
            if (centsString.length() <= 1)
            //if one digit meaning tenths of a dollar
                centsString = centsString + "0";
            dollars = Long.parseLong(dollarsString);
            cents = Long.parseLong(centsString);
            if ((dollars < 0) || (cents < 0) || (cents > 99))
            {
                System.out.println(
                   "Error: Illegal representation of money.");
                System.exit(0);
```

图5-18　（续）

```
            }
        }
    }
    public void readInput()
    {
        System.out.println("Enter amount on a line by itself:");
        Scanner keyboard = new Scanner(System.in);
        String amount = keyboard.nextLine();
        set(amount.trim());
    }
    /**
     Does not go to the next line after outputting money.
    */
    public void writeOutput()
    {
        System.out.print("$" + dollars);
        if (cents < 10)
            System.out.print(".0" + cents);
        else
            System.out.print("." + cents);
    }

    /**
     Returns n times the calling object.
    */
    public Money times(int n)
    {
        Money product = new Money();
        product.cents = n*cents;
        long carryDollars = product.cents/100;
        product.cents = product.cents%100;
        product.dollars = n*dollars + carryDollars;
        return product;
    }

    /**
     Returns the sum of the calling object and the argument.
    */
    public Money add(Money otherAmount)
    {
        Money sum = new Money();
        sum.cents = this.cents + otherAmount.cents;
        long carryDollars = sum.cents/100;
        sum.cents = sum.cents%100;
        sum.dollars = this.dollars+ otherAmount.dollars + carryDollars;
        return sum;
    }
}
```

由于在美元符号和数字之间可能有空格存在，所以使用了nextLine，而不是next。

图5-18 （续）

当然，要实现Money类，必须选择某种数据表示形式。我们希望将钱款数额表示为精确的量值，因此，会使用整数类型。但是，int类型不能表示非常大的数字，因此无法用它来简单地表示很大的钱款数额。所以，会使用long类型。$3500.36这样的钱款数额要用两个存储在long类型实例变量中的整数——在这个例子中是3500和36——来表示。当然，负的钱款数额也是有意义的，最终的专业级Money类应该允许存在负钱款数额，但我们只是希望有一个相当简单的例子以供学习之用，所以我们这个例子将仅限于使用非负钱款数。

注意被重载了的方法名set。4个名为set的方法名使程序员可以用任何方便的形式来设置钱款数额。程序员可以为不包含美分的美元数使用单个的整数值，可以使用单个的double类型值，也可以使用另一个Money类型对象，或者一个字符串，比如"$9.98"或"9.98"。程序员不用也用不着操心Money类中使用了哪些实例变量。相反，程序员考虑的应该是钱款数额，而不是任何实例变量。

现在来看看set方法定义中的一些细节。带有一个long类型参数的set方法是很简单的。

带有一个double类型参数的set方法将double值转换一个用美分总数来表示钱款数额的值。转换是通过下列代码实现的：

```
long allCents = Math.round(amount*100);
```

方法Math.round略去了分币的小数部分。如这个例子所示，当实参为double类型时，这个方法会返回一个long类型的值。然后再用整除运算符/和%将分币数转换成美元和美分。

带有一个Money类型参数的set方法也是很简单的，但应该注意到一个很重要的问题。在Money类的类定义内部可以直接访问参数的实例变量，如otherObject.dollars。在类定义中（比如在Money的定义中），可以直接访问属于那个类的任意对象的实例变量。

带有一个String类型参数的set方法是一个字符串处理的练习。它会将"$12.75"或"12.75"这样的字符串转换成两个整数，比如12和75。首先，要查看字符串中的第一个字符是否是美元符号。检查是按下列方式进行的：

```
if (amountString.charAt(0) == '$')
    amountString = amountString.substring(1);
```

字符串方法调用amountString.charAt(0)返回了字符串amountString中的第一个字符。如果这个字符是'$'，那么，从索引位置0到最后一个索引之间的字符串就会被从索引号1到最后一个索引之间的字符串取代，从而有效地删除第一个字符。这是通过下列方式完成的：

```
amountString = amountString.substring(1);
```

在字符'$'和美元数值之间可能还会有一个或多个空格。可以通过下列对String方法trim的调用删除这些空格：

```
amountString = amountString.trim();
```

然后，对小数点进行定位，并从小数点处将字符串断开，字符串就被分成美元子串和美分子串。小数点的位置按照下列方式存储在变量pointLocation中：

```
int pointLocation = amountString.indexOf(".");
```

按下列方式恢复美元和美分子串：

```
dollarsString =
    amountString.substring(0, pointLocation);
centsString =
    amountString.substring(pointLocation + 1);
```

你可能需要回顾图2-7中对substring方法的描述。

最后，用包装类Long中的静态方法parseLong将美元和美分子字符串转换成long类型的值：

```
dollars = Long.parseLong(dollarsString);
cents = Long.parseLong(centsString);
```

如果不太熟悉方法parseLong，可以参考5.2.4节。

方法times用来将钱款数乘以一个整数。方法add用来对两个Money类型的对象相加。例如，如果m1和m2都是表示数值$2.00的Money类型对象，m1.times(3)就会返回一个表示数值$6.00的Money类型对象，而m1.add(m2)会返回一个表示数值$4.00的Money类型对象。

要理解方法times和add的定义，就要记住1美元中有100美分，因此，如果product.cents的值为100或更多，下列代码就会将变量carryDollars的值设置为相应数额美分中的完整美元数。

```
long carryDollars = product.cents/100;
```

去掉相应数额美元之后，剩下的美分数由下列语句给出：

```
product.cents%100
```

图5-19给出了一个Money类的演示程序。

```
public class MoneyDemo
{
    public static void main(String[] args)
    {
        Money start = new Money();
        Money goal = new Money();

        System.out.println("Enter your current savings:");
        start.readInput();

        goal = start.times(2);
        System.out.print(
            "If you double that, you will have ");
        goal.writeOutput();
        System.out.println(", or better yet:");
        goal = start.add(goal);
        System.out.println(
            "If you triple that original amount, you will have:");
        goal.writeOutput();
        System.out.println();

        System.out.println("Remember: A penny saved");
        System.out.println("is a penny earned.");
    }
}
```

需要结束这一行，因为writeOutput并没有将行结束。

屏幕对话示例

```
Enter your current savings :
Enter amount on a line by itself:
$500.99
If you double that, you will have $1001.98, or better yet:
If you triple that original amount, you will have
$1502.97
Remember: A penny saved
is a penny earned.
```

图5-19　Money类的使用

自测题

31. 重新编写图5-18中的方法add，使其不再使用参数this。
32. 在图5-18中，带有一个String参数的set方法不允许字符串中存在前导及尾部空格。重新编写这个方法，使其忽略前导及尾部的空白。例如，这个方法应该允许将"$5.43"作为它的实参。

5.5 构造器

要事优先。

——俗语

创建一个类对象时，通常希望执行一些初始化动作，比如为一些实例变量赋值。构造器是用来执行这类初始化动作的特殊方法。本节将介绍如何定义及使用构造器。

定义构造器

到目前为止，已经用下面的形式创建过新对象了：

```
Pet goodScout = new Pet();
```

（Pet类是在图5-17中定义的，但我们的问题与这个特定类的细节无关。）对于到目前为止所见过的类来说，这样会创建一个实例变量没有被初始化的对象（或者实例变量有默认的初始值，但默认值可能不是你所期望的值）。你可能希望在创建对象时，将部分或所有实例变量都自动地初始化为指定的值。可以通过一个名为构造器的特殊类型的方法来实现这一点。

构造器（constructor）是在创建一个新对象时调用的方法。它可以执行写入其定义中的任何动作，但构造器应该是执行初始化动作的，比如对实例变量值的初始化。构造器的作用与图5-17所示Pet类定义中的set方法完全相同。但与set方法不同的是，每当用new运算符创建一个对象时，基本上都会自动调用构造器。

构造器的一个乍看起来可能很奇怪的特性就是，每个构造器都和它的类同名。因此，如果类名为Pet，那么构造器也会被称为Pet；如果类名为Species，构造器也会被称为Species。

例如，在图5-20中重新编写了宠物记录的类定义，使其包含了构造器方法。就像set方法一样，构造器通常都会被重载，这样就会有多个构造器定义，每个定义都有不同数量或不同类型的参数。图5-20和图5-17类似，但它们之间大部分区别都是很重要的。下面来看看它们之间的这些区别。

（1）因为每个类都应该有不同的名字，所以在图5-20中将类名从Pet改为PetRecord。同样，由于方法main只是一个不再需要的演示程序，所以在图5-20中将其删除了。

（2）在图5-20中添加了一些名为PetRecord的构造器方法。注意，这些构造器的头部没有像set方法那样包含单词void。定义一个构造器时，不需要指定任何返回类型，甚至不需要写上void来取代返回值类型。这些构造器与set方法非常相似。但是，与某些set方法不同的是，虽然可能不是每个实例变量都有相应的实参，构造器还是为所有的实例变量都赋了值。即使没有为某些实例变量赋值，也可以对构造器进行编译，但在定义构造器时为所有的实例变量赋值是一种常规。构造器和set方法的使用方式是相关但不同的。

(3) 在图5-20中还添加了一个没有参数的构造器PetRecord。无论何时，只要至少定义了一个构造器，就一定要包含带有零个参数的构造器。这样的构造器被称为默认构造器(default constructor)。

```
/**
 Class for basic pet records: name, age, and weight.
*/
public class PetRecord
{
    private String name;
    private int age;//in years
    private double weight;//in pounds

    public void writeOutput()
    {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age + " years");
        System.out.println("Weight: " + weight + " pounds");
    }
    public PetRecord(String initialName, int initialAge,
                                         double initialWeight)
    {
        name = initialName;
        if ((initialAge < 0) || (initialWeight < 0))
        {
            System.out.println("Error: Negative age or weight.");
            System.exit(0);
        }
        else
        {
            age = initialAge;
            weight = initialWeight;
        }
    }
    public void set(String newName, int newAge, double newWeight)
    {
        name = newName;
        if ((newAge < 0) || (newWeight < 0))
        {
            System.out.println("Error: Negative age or weight.");
            System.exit(0);
        }
        else
        {
            age = newAge;
            weight = newWeight;
        }
    }
    public PetRecord(String initialName)
    {
        name = initialName;
```

只有在用new创建对象时才会调用构造器。要修改现有对象，就需要一个或多个这样的set方法。

图5-20 带有构造器的PetRecord类

```
        age = 0;
        weight = 0;
    }

    public void set(String newName)
    {
        name = newName; //age and weight are unchanged.
    }

    public PetRecord(int initialAge)
    {
        name = "No name yet.";
        weight = 0;
        if (initialAge < 0)
        {
            System.out.println("Error: Negative age.");
            System.exit(0);
        }
        else
            age = initialAge;
    }

    public void set(int newAge)
    {
        if (newAge < 0)
        {
            System.out.println("Error: Negative age.");
            System.exit(0);
        }
        else
            age = newAge;
        //name and weight are unchanged.
    }

    public PetRecord(double initialWeight)
    {
        name = "No name yet";
        age = 0;
        if (initialWeight < 0)
        {
            System.out.println("Error: Negative weight.");
            System.exit(0);
        }
        else
            weight = initialWeight;
    }
    public void set(double newWeight)
    {
        if (newWeight < 0)
        {
            System.out.println("Error: Negative weight.");
            System.exit(0);
```

图5-20　（续）

```
        }
        else
            weight = newWeight; //name and age are unchanged.
    }

    public PetRecord()  ◄──────── 默认构造器
    {
        name = "No name yet.";
        age = 0;
        weight = 0;
    }

    public String getName()
    {
        return name;
    }

    public int getAge()
    {
        return age;
    }

    public double getWeight()
    {
        return weight;
    }
}
```

图5-20 （续）

用new创建一个对象时就会调用构造器。在图5-17的程序中已经通过下面这样的语句使用过构造器了：

```
Pet myDog = new Pet();
```

这行代码将myDog定义为一个Pet类对象的名字，然后创建了一个新的Pet类对象。创建新对象的那部分是：

```
new Pet()
```

Pet()部分是对Pet类构造器的调用。由于这个构造器没有实参，所以圆括号内是空的。

通过图5-17中Pet类的定义，会发现类定义中根本就没有构造器的定义。无论什么时候，如果类定义中没有构造器定义，Java都会自动地创建一个默认的构造器，即一个带有零个参数的构造器。这个自动创建的构造器本质上什么都没做，但它可以让你创建类的对象。但是，一旦向类中至少添加了一个构造器定义，就要负责它所有的构造器了。即一旦向一个类中至少添加了一个构造器，Java就不会自动创建任何构造器了。因此，在图5-20中，在带有构造器的PetRecord类中，我们很小心地包含了一个没有参数的构造器作为默认构造器。图5-21给出了一个完整的程序，用PetRecord类说明了构造器的使用。

```
import java.util.*;

public class PetRecordDemo
{
    public static void main(String[] args)
    {
        PetRecord usersPet = new PetRecord("Jane Doe");
        System.out.println("My records on your pet are inaccurate.");
        System.out.println("Here is what they currently say:");
        usersPet.writeOutput();

        Scanner keyboard = new Scanner(System.in);
        System.out.println("Please enter the correct pet name:");
        String correctName = keyboard.nextLine();

        System.out.println("Please enter the correct pet age:");
        int correctAge = keyboard.nextInt();
        System.out.println("Please enter the correct pet weight:");
        double correctWeight = keyboard.nextDouble();
        usersPet.set(correctName, correctAge, correctWeight);
        System.out.println("My updated records now say:");
        usersPet.writeOutput();
    }
}
```

屏幕对话示例

```
My records on your pet are inaccurate.
Here is what they currently say:
Name: Jane Doe
Age: 0
Weight: 0.0 pounds
Please enter the correct pet name:
Moon Child
Please enter the correct pet age:
5
Please enter the correct pet weight:
24.5
My updated records now say:
Name : Moon Child
Age: 5
Weight: 24.5 pounds
```

图5-21　构造器和set方法的使用

用运算符new创建一个新对象时，通常都应该在运算符new后面包含一个对构造器的调用。和所有的函数调用一样，要在（与类名相同的）构造器名后面的圆括号中列出所有实参。比如，假设你想用new来创建图5-20中定义的PetRecord类的一个新对象。就要按如下形式操作：

```
PetRecord fish = new PetRecord("Wanda", 2, 0.25);
```

PetRecord("Wanda", 2, 0.25)这部分是调用PetRecord类中带有3个实参的构造器：一个实

参是String类型的，一个实参是int类型的，最后一个实参是double类型的。这条语句会创建一个新的对象，用来表示一个名为Wanda的、2岁大、重0.25磅的宠物。

再如：

```
PetRecord newBorn = new PetRecord();
```

这条语句创建了PetRecord类的一个新对象，并调用了默认的构造器（即带有零个参数的构造器）。通过图5-20中PetRecord类的定义，你就会发现带有零个参数的构造器将对象名设为"No name yet"，并将实例变量age和weight都设置为0。（当然，新出生宠物的重量也不会为零。0这个值只是作为一个占位符使用，直到确定了宠物的实际重量为止，但是无论如何，这都是生物学，而不是计算机科学的问题。）

只有在用运算符new创建一个新对象时才会调用构造器。下面对PetRecord类对象的调用是非法的：

```
newBorn.PetRecord("Fang", 1, 150.0);   //Invalid!
```

由于你无法在对象创建之后调用它的构造器，所以就需要其他一些方法来修改对象实例变量的值。这就是图5-20中set方法的作用。因此，我们会用下面的set调用来代替前面对构造方法PetRecord的非法调用：

```
newBorn.set("Fang", 1, 150.0);
```

不一定要将这些方法命名为set；可以使用任何便于使用的方法名。例如，你可能更喜欢将它们命名为reset或giveNewValues。但传统上会将这些方法命名为set或其他包含"set"的名字。

快速参考：构造器

构造器是在用new创建类对象时调用的方法。构造器是用来初始化对象的。构造器必须与它所属的类同名。构造器的实参是在类名后的圆括号中给出的。

举例：

```
PetRecord myDog = new PetRecord("Fido", 2, 4,5),
          yourDog = new PetRecord("Cha Cha", 3, 2.3);
```

构造器的定义与任何其他函数都是类似，只是在它的方法头部没有返回值类型，甚至也没有void。构造器定义的实例参见图5-20。

记住：构造器返回一个引用

可以认为new PetRecord()这样的构造器调用返回了一个对象的引用，即认为它返回了对象的内存地址。

举例：

如果PetRecord是一个类，那么PetRecord就是这个类的构造器名，new PetRecord()就是对PetRecord类的构造器的调用。你可以认为这个构造器调用返回了对PetRecord类对象的引用（返回了它的内存地址）。如果pet是一个PetRecord类型的变量，那么下列语句就将这个引用赋给了变量pet：

```
pet = new PetRecord();
```

图5-22说明了这是如何工作的。

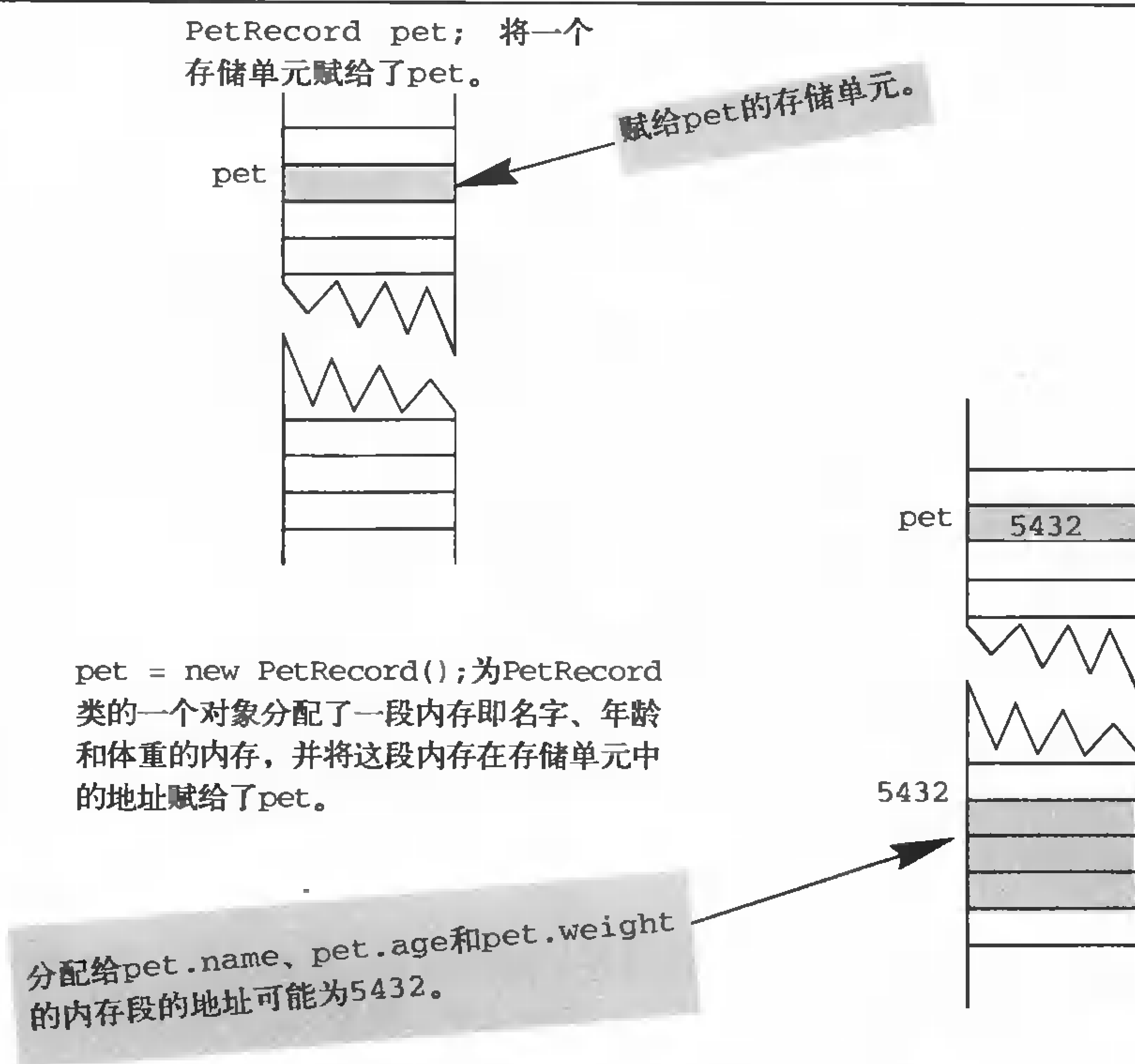


图5-22 返回引用的构造器

常见问题：如何在类图中编写构造器？

类图中不需要包含类中所有的方法。类图是一个设计工具，只要对手头的设计任务来说够用即可。通常不会在类图中列出构造器，这是因为通常构造器是都需要的，而且构造器的基本功能都是一样的。

● 编程提示：可以在构造器中使用其他方法

构造器是用来创建对象的方法，因此它就是对象调用的第一个方法。但是，这不应该妨碍在构造器的定义中使用同一个类中的其他方法。例如，图5-20的PetRecord类定义中所有的构造器都可以用set方法重新编写。考虑类定义中的下列构造器定义：

```
public PetRecord(String initialName, int initialAge,
                                     double initialWeight)
{
    name = Name;
    if ((initialAge < 0) || (initialWeight < 0))
    {
        System.out.println("Error: Negative age or weight.");
        System.exit(0);
    }
    else
    {
```

```
        age = initialAge;
        weight = initialWeight;
    }
}
```

如果愿意，也可以用下列等效的定义来取代它：

```
public PetRecord(String initialName, int initialAge,
                                        double initialWeight)
{
    set(initialName, initialAge, initialWeight);
}
```

类定义中其他构造器也可以用方法set重新编写。

很多程序员都愿意采用第二种使用了set的定义，因为它当然要短小一些。有些程序员可能愿意采用第一种版本，因为它避免了可能由set实参的次序造成的混乱，也避免了调用其他方法的开销。两种版本都是可以的，你可以使用你的教员或指导老师喜欢采用的版本。

▲ 易犯错误：省略默认构造器

假设要从图5-20的PetRecord类定义中将带有0个参数的构造器省略，即假设省略了下列构造器定义：

```
public PetRecord()
{
    name = "No name yet";
    age = 0;
    weight = 0;
}
```

省略了这个构造器之后，下列代码就是非法的，并会生成一条错误消息：

```
PetRecord heinz57 = new PetRecord();
```

你可能会指出：如果没有定义任何构造器，Java会自动提供一个默认的构造器，因此，这条语句应该还是合法的。但是，情况要比这稍微复杂一些。如果类定义中没有构造器，Java就会自动提供一个默认的构造器。但是，如果类定义中至少包含了一个构造器定义，Java就不会为提供任何构造器了。一旦你开始定义构造器，就要对构造器完全负责，而且除了你定义的构造器之外，Java不会再生成其他构造器。

类通常都会被一次次地复用，你可能迟早会创建一个没有指定参数的新对象，就像在下列语句中一样：

```
PetRecord heinz57 = new PetRecord();
```

如果遵循这一原则，在你定义的每个类中都包含一个默认的构造器定义，就能避免很多问题。 △

快速参考：默认构造器

没有任何参数的构造器被称为**默认构造器**。你定义的大多数类都应该包含一个默认构造器。

▲ 易犯错误：很多包装类都没有默认构造器

包装类Integer、Double、Character、Boolean、Byte、Short、Long和Float都没有默认

构造器。创建这些类的新对象时，必须像下面的例子一样为实参赋一个初始值：

```
Character myMark = new Character('Z');
```

这是因为构造器的实参是设置这些包装类的实例变量的唯一途径。 △

自测题

33. 如果有一个名为Student的类，应该为这个类的构造器使用什么名字呢？
34. 在定义构造器时，要为返回值指定什么类型呢？基本类型？类类型？还是void？
35. 什么是默认构造器？
36. Java中的每个类都会自动拥有一个默认构造器吗？如果不是，那么什么时候Java会自动提供一个默认的构造器，什么时候不会提供？

5.6 再论信息隐藏

对理解本书其余大部分内容来说，本节的内容不是必需的。如果你愿意，可以很安全地将本节内容推迟到你对类更熟悉时再读。本节讨论了在定义某些种类的类时可能会出现的一个微妙问题。如果你现在不想阅读本节的内容，又想避免本节讨论的问题，只要注意到下面这一点就行了：如果类中的每个实例变量都是基本类型（比如int、double、char和boolean）或String类型，就不会遇到这个问题。这样，不需触及这个问题就可以定义很多类了。

▲ 易犯错误：隐私泄露

类中可以有任意类型的实例变量，包括任意的类类型。有时，这是一种自然且有用的。但是，使用类类型的实例变量会引入一个需要特别关注的问题。问题出在类类型的变量中包含的是内存中存储对象的内存地址。例如，假设goodGuy和badGuy都是图5-20中定义的PetRecord类型的变量。现在，假设goodGuy命名了某个对象，而程序执行了下列赋值语句：

```
badGuy = goodGuy;
```

执行了这条赋值语句之后，badGuy和goodGuy就是同一个对象的两个名字了。因此，如果修改了badGuy，也就修改了goodGuy。为这条赋值语句多增加一点儿上下文，来看看它有什么含义：

```
PetRecord goodGuy = new PetRecord();
goodGuy.set("Faithful Guard Dog", 5, 75);
PetRecord badGuy;
badGuy = goodGuy;
badGuy.set("Dominion Spy", 1200, 500);
goodGuy.writeOutput();
```

因为badGuy和goodGuy命名的是同一个对象，所以，这段代码会生成下列输出：

```
Name: Dominion Spy
Age: 1200 years
Weight: 500.0 pounds
```

由于goodGuy和badGuy命名的是同一个对象，所以对badGuy的修改也会改变goodGuy。对实例变量来说也会发生同样的事情，并会引发一些微妙的问题。我们来看一个例子。

图5-23包含了一个名为CadetClass的类的定义，这个类是一个新手程序员作为家庭练习编写的。这个类没有很多的方法，但这不是什么问题。问题在于这个新手错误地认为实例变量pet命名的数据

无法被任何使用PetRecord类的程序修改。这是新手很容易犯的错误。毕竟，把实例变量pet设为私有的了，这样就无法用名字来访问它了。而且为了更安全，没有包含任何可以对私有实例变量pet进行修改的设置方法。这位新手让任何人都可以通过使用公有访问方法getPet看到对象pet的值，但却认为没有程序员能够改变"Faithful Guard Dog"。

```
/**
 Example of a class that does NOT correctly
 hide its private instance variable.
*/
public class CadetClass
{
    private PetRecord pet;

    public CadetClass()
    {
        pet =
            new PetRecord("Faithful Guard Dog", 5, 75);
    }

    public void writeOutput()
    {
        System.out.println("Here's the pet:");
        pet.writeOutput();
    }

    public PetRecord getPet()
    {
        return pet;
    }
}
```

实际的类中应该有更多的方法，但这儿这些就是演示所需的所有内容了。

图5-23　一个不安全的类

图5-24中的程序修改了私有实例变量pet命名的对象中的实例变量的值！这怎么可能呢？问题在于，类类型的变量中存储的是一个内存地址，这就意味着你可以用赋值运算符产生同一个对象的两个名字。这就是编写图5-24程序的黑客所作的。黑客用访问方法getPet返回了私有实例变量pet的值。但那个值是存储在变量badGuy中的一个内存地址。因此，badGuy就是pet的另一个名字。黑客无法使用私有名pet，但可以使用等价的名字badGuy。黑客要做的所有工作就是用名字badGuy来调用PetRecord类的set方法，而且因为badGuy是pet所命名对象的另一个名字，所以，黑客已经修改了私有实例变量pet命名的对象了。

怎样编写类定义才能避免这个问题呢？看起来好像不可能有真正安全的类类型私有实例变量了，黑客总是可以找到一种办法以得到这个实例变量或者这个实例变量的一个引用。但是，至少有两种方法可以解决这个问题：简单的办法和较复杂但更好的办法。

解决这个问题的简单办法就是只使用基本类型或String类型的实例变量。String类型没有可以改变String对象数据的设置方法，因此黑客的那套方法对它们来说是没用的。基本类型不是类类型，因此黑客的那套方法对它们也是没用的。在本书中采用的就是这种简单的解决办法。（如果在某些类类型中没有可以改变对象的方法，那么，除了String之外，也可以拥有这些类类型的实例变量。但是这里经常会碰到的这种类类型只有String类。）

```
/**
 Toy program to demonstrate how a programmer can access and
 change private data in an object of the class CadetClass.
*/
public class Hacker
{
    public static void main(String[] args)
    {
        CadetClass starFleetOfficer = new CadetClass();
        System.out.println("starFleetOfficer contains:");
        starFleetOfficer.writeOutput();
        PetRecord badGuy;
        badGuy = starFleetOfficer.getPet();
        badGuy.set("Dominion Spy", 1200, 500);
        System.out.println("Looks like a security breach:");
        System.out.println("starFleetOfficer now contains:");
        starFleetOfficer.writeOutput();
        System.out.println("The pet wasn't so private!");
    }
}
```

屏幕输出

```
starFleetOffier contains:
Here's the pet:
Name: Faithful Guard Dog
Age: 5 years
Weight: 75.0 pounds
Looks like a security breach:
starFleetOfficer now contains:
Here's the pet:
Name: Dominion Spy
Age: 1200 years
Weight: 500.0 pounds
The pet wasn't so private!
```

这个程序修改了一个由对象starFleetOfficer的一个私有实例变量命名的对象。

图5-24　修改一个定义拙劣的类中的私有数据

更复杂的解决方法也是更好的解决方法，但这已经超出了本书的讨论范围。有一些方法可以生成对象的精确副本。这些精确副本被称为**克隆**（clone）。这种较复杂的解决方案不会返回一个由可能不安全的类类型私有实例变量命名的对象，而是会返回对象的一个克隆。这样，黑客可以对克隆做任何他想做的事情，而不会影响到私有数据。附录H对克隆进行了简要的介绍。逐渐熟悉了类的概念之后，可能就想看看相关介绍了。

不要得到这样一种印象：使用类类型的实例变量不是什么好主意。它们是很自然也很有用的。但是，要想有效地处理它们，还需要技巧和细心。就像对待脑部手术一样对待它们：如果你需要它，它是非常非常有用的，但是除非你知道你在做什么，否则就不要去尝试它。　△

自测题

37. 给出可以在图5-23的CadetClass类定义中使用的3个访问方法的定义，而不只是给出单个访问方法

getPet的定义。这些新的访问方法不会引发易犯错误小节“隐私泄露”中描述的问题。它们会返回一个CadetClass类对象中所有的数据，但如果对象中带有可以对内容进行修改的设置方法，它们不会返回这样的对象。一个方法会返回宠物的名字，一个方法会返回宠物的年龄，一个方法会返回宠物的体重。前面“易犯错误”小节中给出了两种避免这个问题的方法。这道题则给出了第3种避免这个问题的方法。

5.7 包

从我的书斋里把那些我看得比一个公国更宝贵的书给我带来。

——威廉·莎士比亚，《暴风雨》

包是组织和命名一组相关类的方法，这样，不需要将所有这些类和你的程序放在同一个目录中，就可以将这组类作为一个类库在任意程序中使用了。尽管这是很重要也很有用的，然而，在本书的其他资料中并没有用到这里讲述的有关包的内容。因此，在阅读本书的过程中，你可以随时来学习本节的内容。

要理解这部分内容，你需要了解目录（在某些操作系统中被称为文件夹）的相关知识，需要了解目录（文件夹）的路径名，以及操作系统是如何使用路径（环境）变量的。如果不了解目录（文件夹）、路径名和路径（环境）变量，应该跳过本节，直到对这些问题有了一些了解为止。这些并不是Java要讨论的话题，它们是操作系统的一部分，有关细节取决于你所使用的特定操作系统。如果你能弄清楚如何设置路径变量，你所了解的路径变量知识就足够理解本节的内容了。

5.7.1 包及其导入

包（package）只是一些被组织起来放在一个目录中、并被赋予了一个包名的类的集合。包中的每个类都放在一个单独的文件中，文件名与类名相同。唯一的区别是每个文件的起始处都会包含下列代码行：

```
package Package_Name;
```

在这行代码之前可能会有一些空行或注释，但在它之前不能有任何其他东西。*Package_ Name* 通常全部由小写字母组成，并由点分隔。例如，如果general.utilities是包的名字，那么包中每个文件都应该将下列代码行放在文件的起始处：

```
package general.utilities;
```

在包含程序或类定义的文件起始处放置一条适当的import语句，这些程序或类定义就可以使用包中所有的类了。即使这些程序或类定义与包中的类不在同一个目录中也是如此。例如，如果想使用general.utilities包中的类，就应该将下列代码放在所编写文件的起始处：

```
import general.utilities.*;
```

5.7.2 包名与目录

包名不是一个任意的标识符。它会告诉编译器到哪里去找包中的类。实际上，包名告诉编译器的是包含包中那些类的目录的路径名。

要找到一个包的目录，Java需要包名以及在类路径变量值中列出的目录。

类路径变量（class path variable）值会告诉Java从哪儿开始搜索包，因此先来讨论它。类路径变量不是一个Java变量，而是操作系统的一部分，包含的是一系列目录的路径名。Java要搜寻一个包时，就会从这些目录中开始搜索。将这些目录称为**类路径基本目录**（class path base directory）。后面会介绍Java是如何使用这些类路径基本目录的，以及包是如何命名的，然后还会给出一些与如何设置类路径变量有关的信息。

快速参考：包

包是一些类的集合，这些类被组织到一个目录中并被赋予了一个包名。包中的每个类都放在一个独立的文件中，文件名与类名相同。包中的每个文件都必须将下列行作为文件的第一条指令行（即第一个非空、也非注释的行）：

语法：

package *Package_Name*;

举例：

```
package general.utilities;
package java.io;
```

快速参考：import语句

在包含程序或类定义的文件起始处放置一条命名了包的import语句，就可以在任意程序或类定义中使用这个包中所有的类了。程序或类不需要与包中的类位于同一个目录中。

语法：

import *Package_Name*;

举例：

```
import  general.utilities.*;
import java.io.*;
```

结尾处的点和*意味着导入了这个包中所有的类。也可以用类名取代*，只从包中导入单个类。

包的名字说明了包含包中那些类的目录的相对路径名。它假设从类路径基本目录开始，沿着包名给出的子目录路径行进，因此，它是一个相对路径名。例如，下面是一个类路径基本目录（在你的操作系统中，可能会用/来取代\）：

```
\myjavastuff\libraries
```

假设包中的类位于目录下面的目录中：

```
\myjavastuff\libraries\general\utilities
```

在这种情况下，必须将包命名为

```
general.utilities
```

注意，包名会指出，从类路径基本目录开始，应该沿着哪些子目录才能找到包中的那些类。图5-25对此进行了说明。注意，包名不是任意取的，而是从类路径基本目录到包中那些类所要经过的目录列表。无论操作系统在目录路径中使用了哪种符号，类名中点的含义本质上都与\和/相同。

可以通过设置类路径（环境）变量来指定类路径基本目录。设置类路径变量的方式取决于所用操作系统。类环境变量通常是一个全部用大写字母拼写的单词，就像CLASSPATH一样。

你很可能会有一个老的“路径变量”，用来告诉操作系统在哪里能找到javac以及其他命令的代码，这些命令都可以作为单行命令给出。如果你能弄清楚如何设置路径变量，就可以用同样的方法来设置CLASSPATH变量了。

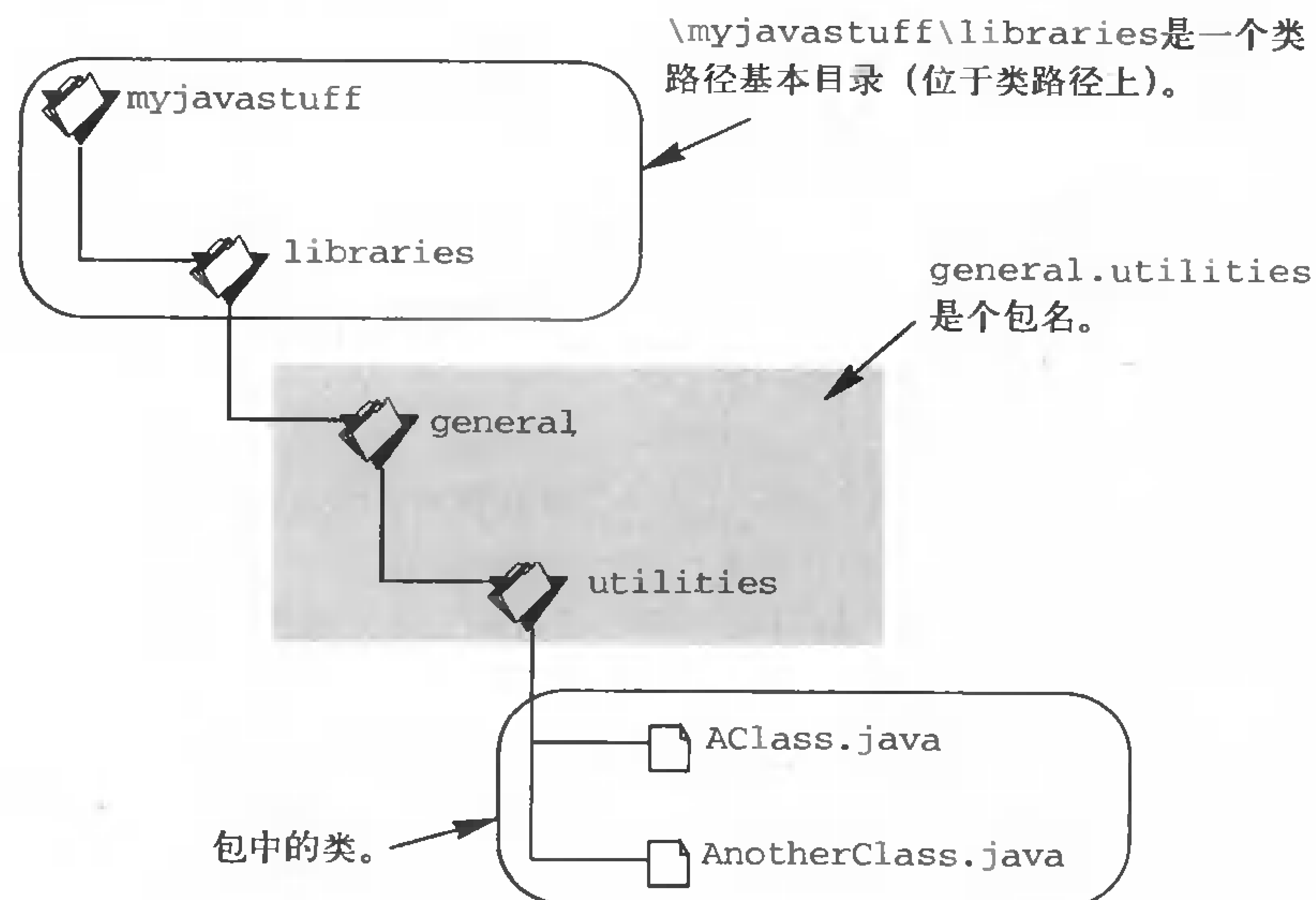


图5-25 包名

如果使用的是UNIX系统，很可能可以用一些与下列命令类似的命令来设置类路径：

```
export CLASSPATH =/myjavastuff/libraries;
```

如果你使用的是Windows操作系统，就可以用控制面板设置或创建一个名为CLASSPATH的环境变量，以此来设置类路径变量。

快速参考：包名

包名必须是包含了包中那些类的目录的路径名，但路径名中用点取代了\或/（无论你的操作系统使用的是哪个符号）。在命名一个包的时候，使用的是一个从类路径（环境）变量中命名的任意一个目录开始的相对路径名。

举例：

```
general.utilities
java.io
```

▲ 易犯错误：在类路径中未包含当前路径

在类路径变量中可以列出多个基本目录，通常用分号对其进行分隔。例如，下面是一个可能的类路径：

```
c:\myjavastuff\libraries;f:\yourjavastuff
```

这就意味着你可以将包目录作为

```
c:\myjavastuff\libraries
```

的子目录，或者

```
f:\yourjavastuff
```

的子目录来创建。

在查找包目录时，Java首先会在

```
c:\myjavastuff\libraries
```

的子目录中寻找。如果在那儿没找到包，它就会到

```
f:\yourjavastuff
```

的子目录中去寻找。

每次设置或者修改类路径变量时，都一定要将当前目录作为一种选择包含在内。**当前目录**（current directory）是你的程序（或其他类）所处的目录。在大多数系统中，当前目录都是用一个点来表示的。因此，应该将下列代码作为你的类路径变量：

```
c:\myjavastuff\libraries;f:\yourjavastuff;.
```

它在末尾添加了当前目录。如果在前面两个基本目录中都没有找到这个包，Java就会在当前目录的子目录中查找，即在你的程序（或你在编译的任何类）的子目录中查找。

如果你希望Java在查看类路径上的其他目录之前查看当前目录，就将当前目录（点）列在最前面。

从类路径变量中省略当前目录不仅会限制使用包的范围，而且会影响那些不使用包的程序；如果编程时不使用包，而是将程序与所有的类都放在同一个目录中（就像我们在本书其余部分所做的那样），那么，除非当前目录在类路径中，否则Java就无法找到那些类。（如果你根本不使用类路径变量，就不会发生这种问题；只有在你决定设置类路径变量时才会出现这样的问题。）△

5.7.3　名字冲突

包是一种很方便的组织和使用类库的方式，但是，使用包还有另一个原因。包有助于处理名字冲突，即它可以帮助处理两个类同名的情况。如果不同的程序员在编写不同包的时候为类使用了相同的名字，就可以通过包名来解决这种含糊不清的情况。

如果一个名为mystuff的包中包含了一个名为CoolClass的类，而另一个名为yourstuff的包中也包含了一个名为CoolClass的类，可以通过使用更完整的名字mystuff.CoolClass和yourstuff.CoolClass，在同一个程序中使用这两个名为CoolClass的类。例如：

```
mystuff.CoolClass object1 = new mystuff.CoolClass();
yourstuff.CoolClass object2 = new yourstuff.CoolClass();
```

因为这种较长的类名中包含了包名，所以，如果像这样列出包名和类名，就不需要导入包了。

自测题

38. 假设你想在编写的程序中使用包mypackages.library1中的类。需要在包含程序的文件起始处放置些什么代码？
39. 假设你想使一个类成为一个名为mypackages.library1的包的成员。需要在包含类定义的文件中添加什么呢？这条语句应该放在文件的什么地方？
40. 可以为一个包任意起一个名字吗，或者对可以用作包名的标识符做什么限制吗？如果存在任何限制，对其进行解释。
41. 在你的操作系统中，将图4-19中的Species类放入一个包中，这样在任意Java程序中，都可以通过包

含一条适当的import语句来使用Species类，而不需要将Species类移到与程序相同的目录（文件夹）中。

5.8 图形编程补充（选读）

现在，你看到了；现在，你看不到了。

——常见的魔术师用语

本节将展示如何向applet添加按钮和图标，以及如何将标签、图标和按钮这样的组件从不可见变成可见的，或者从可见变为不可见。*按钮*（button）只是applet中一个看起来像按钮一样的组件，点击它时，它会做一些事情。创建按钮的方式与创建标签的方式很类似。向applet添加按钮的方式与向applet添加标签的方式相同，但是，按钮也有些新的东西。可以将一个动作与按钮关联起来，这样，点击按钮时，applet就会执行某些动作。*图标*（icon）就是一幅小图片。

● 编程提示：将表现和动作分开编写

在设计及编写一个applet时，一个很好的技巧是：先对applet的外观编程，比如它会显示多少个按钮，以及每个按钮上会写些什么。使applet看起来与你的期望一致之后，就可以继续编写applet所要执行的动作了，比如当用户点击一个按钮时会发生什么事情。通过这种方式可以将一个大型任务分解成两个较小且较易管理的任务。下面将用这种技术来介绍按钮。首先介绍如何向applet中添加按钮。点击这些按钮时，它们什么事情也不会做。了解了如何向applet中添加按钮之后，我们会继续介绍如何对动作进行编程，以响应对按钮的点击。

5.8.1 添加按钮

创建**按钮**对象的方式与创建标签对象的方式相同，但要用JButton类取代JLabel类。例如，下面来自图5-26的代码创建了一个按钮：

```
JButton sunnyButton = new JButton("Sunny");
```

JButton类的构造器实参，在这个例子中就是"Sunny"，是一个在显示按钮时会写在按钮上的字符串。如果去看看图5-26中的applet，就会发现这两个按钮上标记了"Sunny"和"Cloudy"。

向applet的内容面板中添加按钮的方法与添加标签的方法相同。例如，sunnyButton是以下列方式添加到图5-26中applet的内容面板上去的：

```
Container contentPane = getContentPane();
     ...
contentPane.add(sunnyButton);
```

如果点击图5-26中的任一个按钮，什么都不会发生。还需要对这些按钮编程。按钮的编程是通过另一种不同的、名为*事件驱动编程*（event driven programming）的编程技术来实现的，因此，在介绍如何对按钮动作进行编程之前，先简单介绍这种技术。

```
import javax.swing.*;
import java.awt.*;

/**
 Simple demonstration of putting buttons in an Applet.
 These buttons do not do anything. That comes in a later version.
*/
public class PreliminaryButtonDemo extends JApplet
{

    public void init()
    {
        Container contentPane = getContentPane();
        contentPane.setBackground(Color.WHITE);

        contentPane.setLayout(new FlowLayout());

        JButton sunnyButton = new JButton("Sunny");
        contentPane.add(sunnyButton);

        JButton cloudyButton = new JButton("Cloudy");
        contentPane.add(cloudyButton);
    }
}
```

得到的applet

如果用户点击了其中一个按钮，什么事情都不会发生。

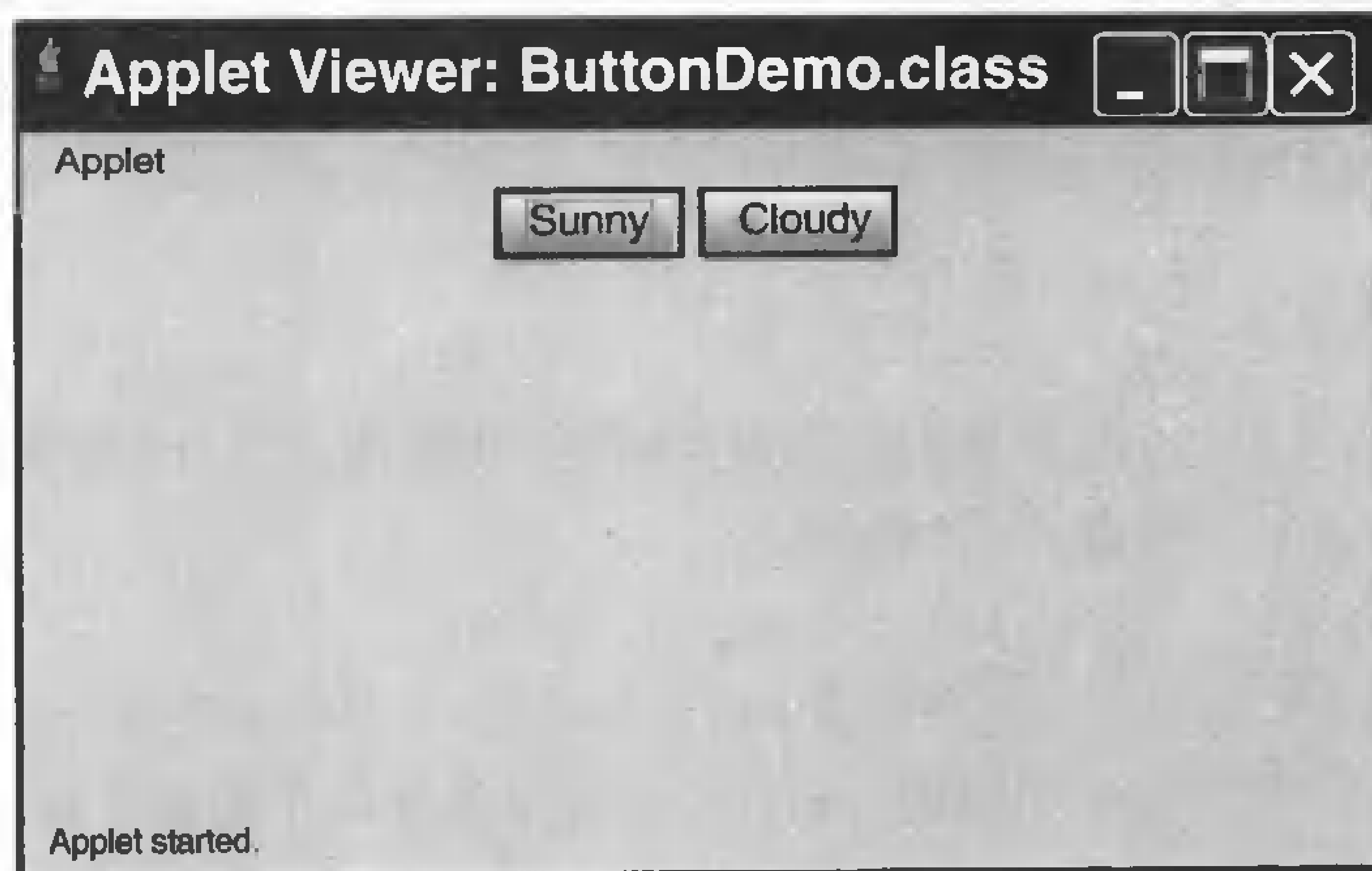


图5-26　向applet中添加一些按钮

5.8.2　事件驱动编程

applet中使用了事件和事件处理程序。**事件**（event）是用来表示某些动作的对象，比如点击一个按钮，或任意它期望能够引发响应的动作。当对象产生一个事件时，比如点击了一个按钮，就称这个按钮**激发**（fire）了事件。在一个applet中，每个可以激发事件的对象，比如一个可能会被点击的按钮，都可以有一个或多个**侦听器对象**（listener object）。作为程序员，要决定哪些对象是可能会激发一个事件给定对象的侦听器对象。比如，如果你点击了一个按

钮，这个动作会激发一个事件，如果这个按钮有一个相关的侦听器对象，这个事件就会被自动发送到这个侦听器对象中。侦听器对象中有一些方法，用来说明侦听器收到各种类型的事件时会发生什么事情。这些处理事件的方法被称为**事件处理程序**（event handler）。程序员负责定义这些事件处理程序方法。图5-27以图表的形式显示了一个事件激发对象，（比如一个按钮）与其事件处理侦听器对象之间的关系。

事件驱动编程与在此之前你见过的其他编程类型有很大区别。在事件驱动编程中，你要创建一些可以激发事件的对象，以及一些对事件进行响应的侦听器对象。在大多数情况下，程序不能确定事件的发生顺序。这个顺序是由各种事件确定的。在运行一个事件驱动程序时，接下来会发生的事情是由下一个事件决定的。

编程示例：一个完整的按钮applet

图5-28的applet似乎和图5-26的applet差不多，但在这个版本中，按钮是可以“工作”的。点击标记为"Sunny"的按钮，背景就会变成蓝色。点击标记为"Cloudy"的按钮，背景就会变成灰色。在5.8.3节中，将解释如何编程才能使这些按钮产生这种行为进行解释。



图5-27 事件的激发与事件侦听器

5.8.3 对按钮进行编程

点击按钮会创建一个事件对象，并会将这个对象发送到另一个（或多个）名为侦听器的对象中。这个过程被称为激发了事件。然后，侦听器会执行一些动作。当我们说事件被“发送”到侦听器对象中去的时候，实际的意思是：将事件对象作为实参，调用了侦听器对象中的一些方法。这种调用是自动发生的。你的applet类定义中通常不会包含对这个方法的调用。但是，applet的类定义确实需要做两件事情：首先，它要为每个按钮指定一个（或多个）会对那个按钮激发的事件进行响应的侦听器对象，这个过程被称为侦听器**注册**（register）；第二，它必须定义一个（或多个）在事件发送到侦听器中时会被调用的方法。

图5-28中的下列代码行将this注册为侦听器，用来接收来自名为sunnyButton的按钮的事件：

```
sunnyButton.addActionListener(this);
```

还有一条类似的语句将this注册为一个负责接收来自cloudyButton按钮的事件。由于实参为this，这条语句就意味着this（ButtonDemo类自身）就是侦听器类。在类定义内部，那个类

的对象就被称为this。因此，ButtonDemo类自己就是ButtonDemo内各个按钮的侦听器类。（更精确地说，ButtonDemo类的每个applet对象就是那个对象中按钮的侦听器。图5-29以图形的形式对此进行了说明。）下面解释一下如何使ButtonDemo这样的类成为按钮激发的事件的侦听器类。

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

/**
 Simple demonstration of putting buttons in an Applet.
 These buttons do something when clicked.
*/
public class ButtonDemo extends JApplet implements ActionListener
{

    public void init()
    {
        Container contentPane = getContentPane();
        contentPane.setBackground(Color.WHITE);

        contentPane.setLayout(new FlowLayout());

        JButton sunnyButton = new JButton("Sunny");
        contentPane.add(sunnyButton);
        sunnyButton.addActionListener(this);

        JButton cloudyButton = new JButton("Cloudy");
        contentPane.add(cloudyButton);
        cloudyButton.addActionListener(this);
    }

    public void actionPerformed(ActionEvent e)
    {
        Container contentPane = getContentPane();

        if (e.getActionCommand().equals("Sunny"))
            contentPane.setBackground(Color.BLUE);
        else if (e.getActionCommand().equals("Cloudy"))
            contentPane.setBackground(Color.GRAY);
        else
            System.out.println("Error in button interface.");
    }
}
```

ActionEvent类的使用需要这条import语句。

除了改变了类名、添加了高亮文本之外，这个applet的代码与图5-26的代码相同。

得到的applet

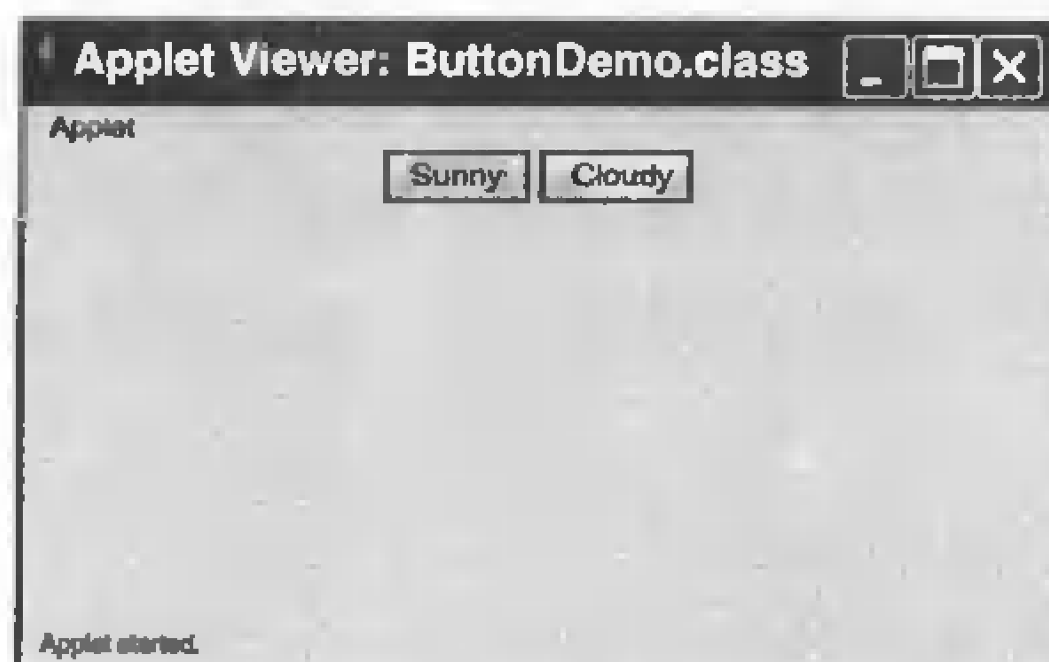
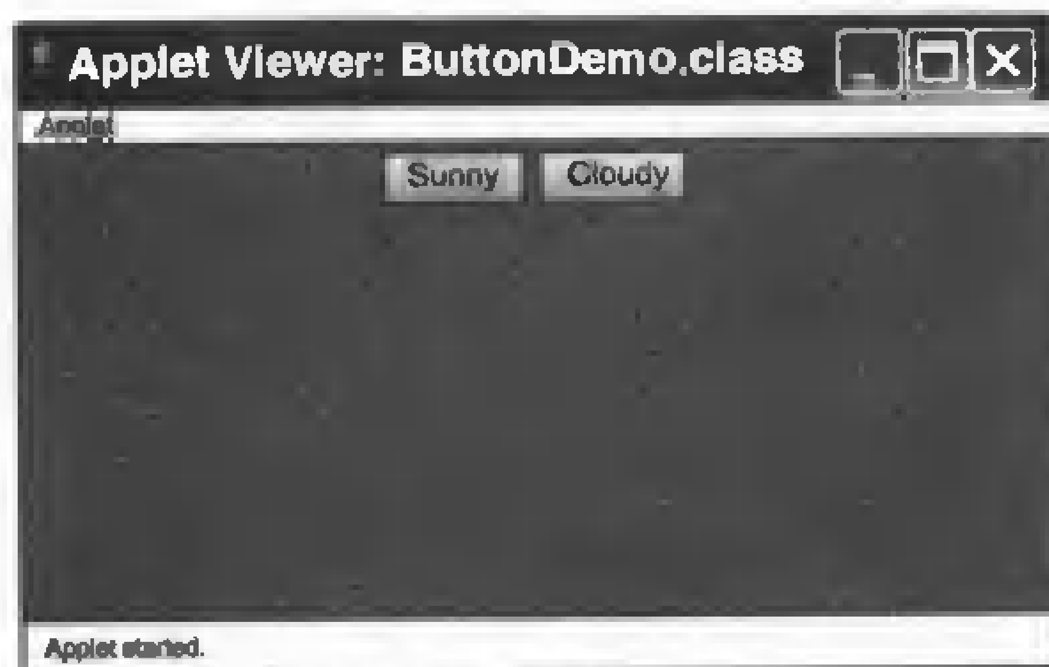


图5-28 向按钮中添加动作

点击Sunny按钮之后得到的applet



点击Cloudy按钮之后得到的applet

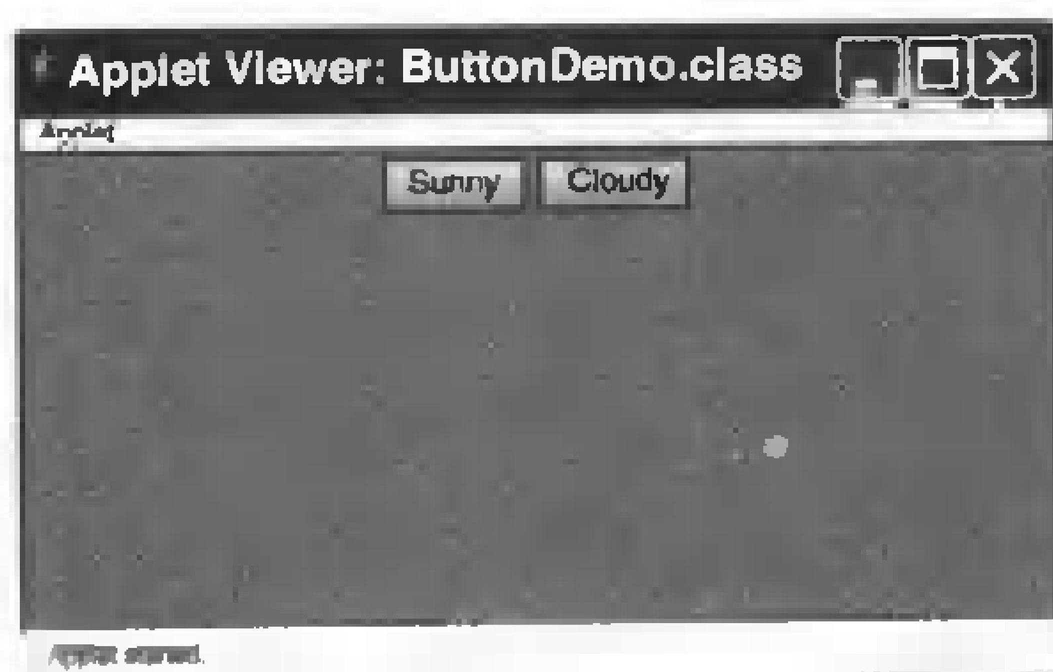


图5-28 （续）

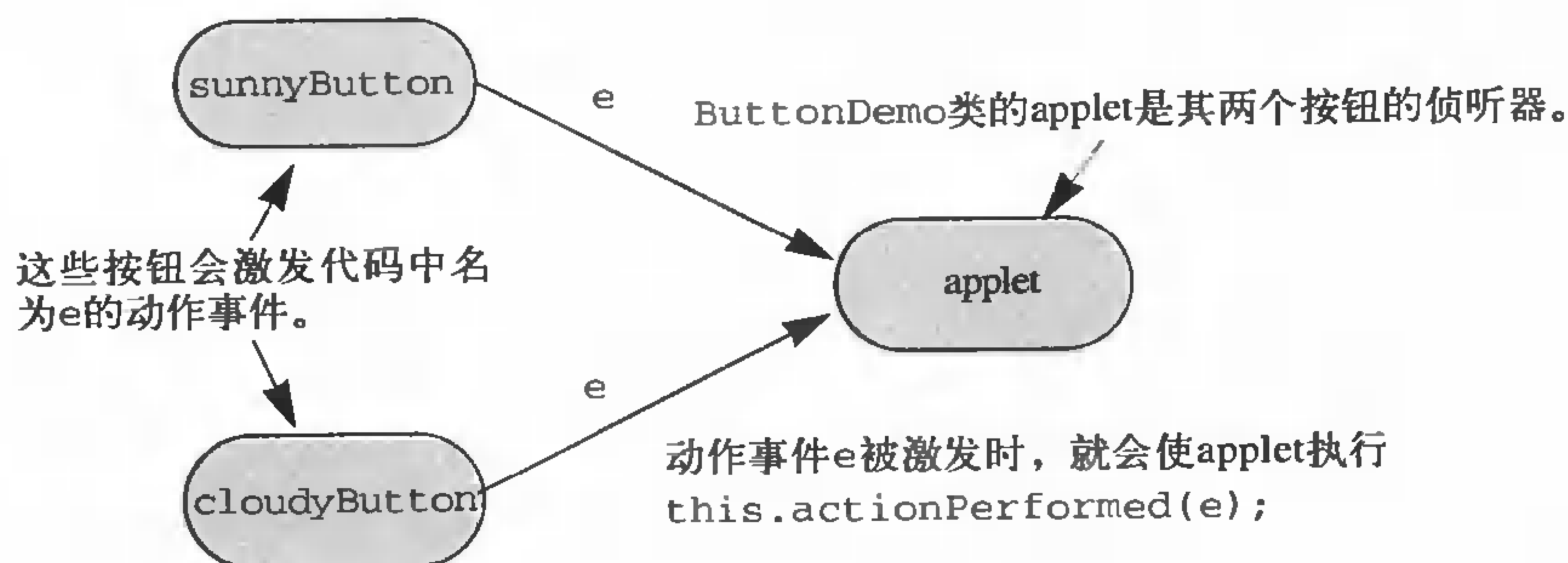


图5-29 按钮与动作侦听器

不同类型的组件需要不同类型的侦听器类来处理它们激发的事件。按钮会激发一些名为**动作事件**（action event）的事件，这些事件是由名为**动作侦听器**（action listener）的侦听程序来处理的。

动作侦听器是ActionListener类型的对象。ActionListener不是类，而是属性。你可以将这项属性赋予你定义的任意类。（这些属性，比如ActionListener，被称为接口（interface），将在第7章详细讨论。）要使一个类成为一个ActionListener，需要做两件事：

(1) 将implements ActionListener添加到类定义的起始处，通常是在第一行的末尾。

(2) 定义一个名为actionPerformed的方法。

在图5-28中，就是用这种方法使一个名为ButtonDemo的applet类成为一个ActionListener的。我们在这里重现了ButtonDemo类定义的轮廓（省略的部分用3个点来表示）：

```
public class ButtonDemo extends JApplet implements ActionListener
```

```
{
                .
                .
                .
    public void actionPerformed(ActionEvent e)
    {
                .
                .
                .
    }
                .
                .
                .
}
```

可以定义一个单独的类，仅用于处理按钮事件，但将applet类ButtonDemo变成一个可以处理按钮事件的ActionListener会更方便一些。假设按钮事件会改变applet，那么改变applet最简单的方式就是使用applet自身内部的方法，所以这样会更方便一些。

如果点击了applet内的一个按钮，就会向那个按钮的动作侦听器发送一个动作事件。但applet自己就是这些按钮的动作侦听器，因此，动作事件会传送到applet对象。动作侦听器收到一个动作事件时，会自动地将事件传递给方法actionPerformed。方法actionPerformed通常是一个分支语句，用来确定要激发哪种类型的动作，然后执行一些相应的动作。图5-28的applet类ButtonDemo中方法actionPerformed的代码如下：

```
public void actionPerformed(ActionEvent e)
{
   Container contentPane = getContentPane();
   if (e.getActionCommand().equals("Sunny"))
       contentPane.setBackground(Color.BLUE);
   else if (e.getActionCommand().equals("Cloudy"))
       contentPane.setBackground(Color.GRAY);
   else
       System.out.println("Error in button interface.");
}
```

在这种情况下，方法actionPerformed需要知道动作事件是来自标记为"Sunny"的按钮，还是来自标记为"Cloudy"的按钮。如果e是通过点击按钮激发的一个动作事件，e.getActionCommand()就会返回写在按钮上的字符串；在这个例子中，就会返回"Sunny"或"Cloudy"。因此，方法actionPerformed需要做的所有工作就是确认e.getActionCommand()是字符串"Sunny"，还是字符串"Cloudy"，然后为相应按钮执行适当的动作。注意，e.getActionCommand()是String类的对象。String类有一个equals方法，可以用来查看e.getActionCommand()是等于"Sunny"，还是等于"Cloudy"（或任何其他字符串）。如果e.getActionCommand()等于*String_Argument*，方法调用

```
e.getActionCommand().equals(String_Argument)
```

就会返回true，否则就返回false。因此，可以通过下列代码来测试e.getActionCommand()等于"Sunny"还是"Cloudy"，并据此来改变内容面板的颜色：

```
if (e.getActionCommand().equals("Sunny"))
    contentPane.setBackground(Color.BLUE);
else if (e.getActionCommand().equals("Cloudy"))
```

```
        contentPane.setBackground(Color.GRAY);
    else
        System.out.println("Error in button interface.");
```

最后一条else子句应该永远都不会执行。写出该子句只是为了让我们在代码中犯了一些不容易被注意到的错误时，能够得到警告。

动作侦听器类的定义需要下列附加的import语句：

```
import java.awt.event.*;
```

快速参考：动作事件与动作侦听器

按钮及其他一些组件会激发ActionEvent类中的事件，这些事件称为**动作事件**（action event），由动作侦听器处理。任何类都可以是动作侦听器类。细节的概要形式如下所示。

（1）通过以下两种方法使某些类（可能是一个applet类）成为一个动作侦听器。

- 向类定义的头部添加下列短语：

  implements ActionListener

- 向类中添加一个名为actionPerformed的方法的定义。

（2）把一个（或多个）ActionListener类对象注册到按钮（或其他可以激发动作事件的组件）。可以用方法addActionListener实现这一点。一个按钮或其他的组件上可以注册多于（或少于）一个的侦听器。

动作侦听器可以是任何类。特别需要指出的是，除了作为动作侦听器之外，类还可以有其他一些功能。

举例（图5-28中有完整的细节）：

```
public class ButtonDemo extends JApplet implements ActionListener
{
          .
          .
          .
   public void init()
   {
          .
          .
          .
       Container contentPane = getContentPane();
          .
          .
          .
       JButton sunnyButton = new JButton("Sunny");
       contentPane.add(sunnyButton);
       sunnyButton.addActionListener(this);
          .
          .
          .
    }
    public void actionPerformed(ActionEvent e)
    {
          .
          .
          .
```

```
        }
    }
```

快速参考：actionPerformed方法

要成为一个动作侦听器，除了其他内容之外，类还必须有一个名为actionPerformed的方法，这个方法有一个ActionEvent类型的参数。这是ActionListener接口所需的唯一一个方法。

语法：

```
public void actionPerformed(ActionEvent e)
{
    Code_for_Actions_Performed
}
```

*Code_for_Actions_Performed*通常是一个取决于e的某些属性的分支语句。分支通常取决于e.getActionCommand()。如果e是一个通过点击按钮激发的事件，e.getActionCommand()就是一个称为**动作命令**（action command）的字符串。动作命令就是写在按钮上的字符串。（可以指定另外一个动作命令，与此相关的细节参见本书姊妹篇《Java程序设计与问题解决：高级篇（第4版）》的第5章。）

举例：

```
public void actionPerformed(ActionEvent e)
{
    Container contentPane = getContentPane();
    if (e.getActionCommand().equals("Sunny"))
        contentPane.setBackground(Color.BLUE);
    else if (e.getActionCommand().equals("Cloudy"))
        contentPane.setBackground(Color.GRAY);
    else
        System.out.println("Error in button interface.");
```

■ Java提示：applet不使用构造器

由于applet通常没有构造器，所以，它们是非典型的类。你要放入applet构造器中的初始化动作将被放在一个特殊方法init中。

自测题

42. 图5-28的init方法中包含了下列代码行：

```
Container contentPane = getContentPane();
```

内容面板一定叫contentPane吗？可以叫做别的什么名字吗？例如：

```
Container insideOfApplet = getContentPane();
```

43. 将事件“发送”给侦听器对象是什么意思？

44. 如果如下所示用标识符buttonEvent取代e，会对图5-28中的程序产生什么影响？

```
public void actionPerformed(ActionEvent buttonEvent)
{
    Container contentPane = getContentPane();
    if(buttonEvent.getActionCommand().equals("Sunny"))
        contentPane.setBackground(Color.BLUE);
    else if (
      buttonEvent.getActionCommand().equals("Cloudy"))
        contentPane.setBackground(Color.GRAY);
```

```
        else
            System.out.println("Error in button interface.");
    }
```

5.8.4 图标

可以向applet添加一些图标。图标只是一个小图片，尽管实际上并不要求它一定很小。图片可以用来描绘任何东西，它是以很多能够显示在计算机屏幕上的格式（比如GIF和JPEG）产生的。基本上任何一种标准格式的图片都可以作为一个图标的素材。Swing可以将这些数字图片文件转换成一个图标，然后你就可以将这个图标添加到标签、按钮或其他组件中去了。标签或按钮上可能只显示一个字符串，或者只显示一个图标，或者两者都显示。

ImageIcon类可以将一个数字图片文件转换成一个Swing图标。例如，图5-30中的下列代码行会将一个名为duke_waving.gif的数字图片文件转换成一个名为dukeIcon的图标：

```
ImageIcon dukeIcon = new ImageIcon("duke_waving.gif");
```

可以用方法setIcon将这个图标添加到一个标签。图5-30中的一个例子如下所示：

```
niceLabel.setIcon(dukeIcon);
```

在图5-30中，标签niceLabel上显示了字符串"Java is fun!"。因此，在这个例子中，标签上既显示了字符串也显示了图标。

```
import javax.swing.*;

public class IconDemo extends JApplet
{
    public void init()
    {
        JLabel niceLabel = new JLabel("Java is fun!");
        ImageIcon dukeIcon = new ImageIcon("duke_waving.gif");
        niceLabel.setIcon(dukeIcon);
        getContentPane().add(niceLabel);
    }
}
```

得到的applet[①]

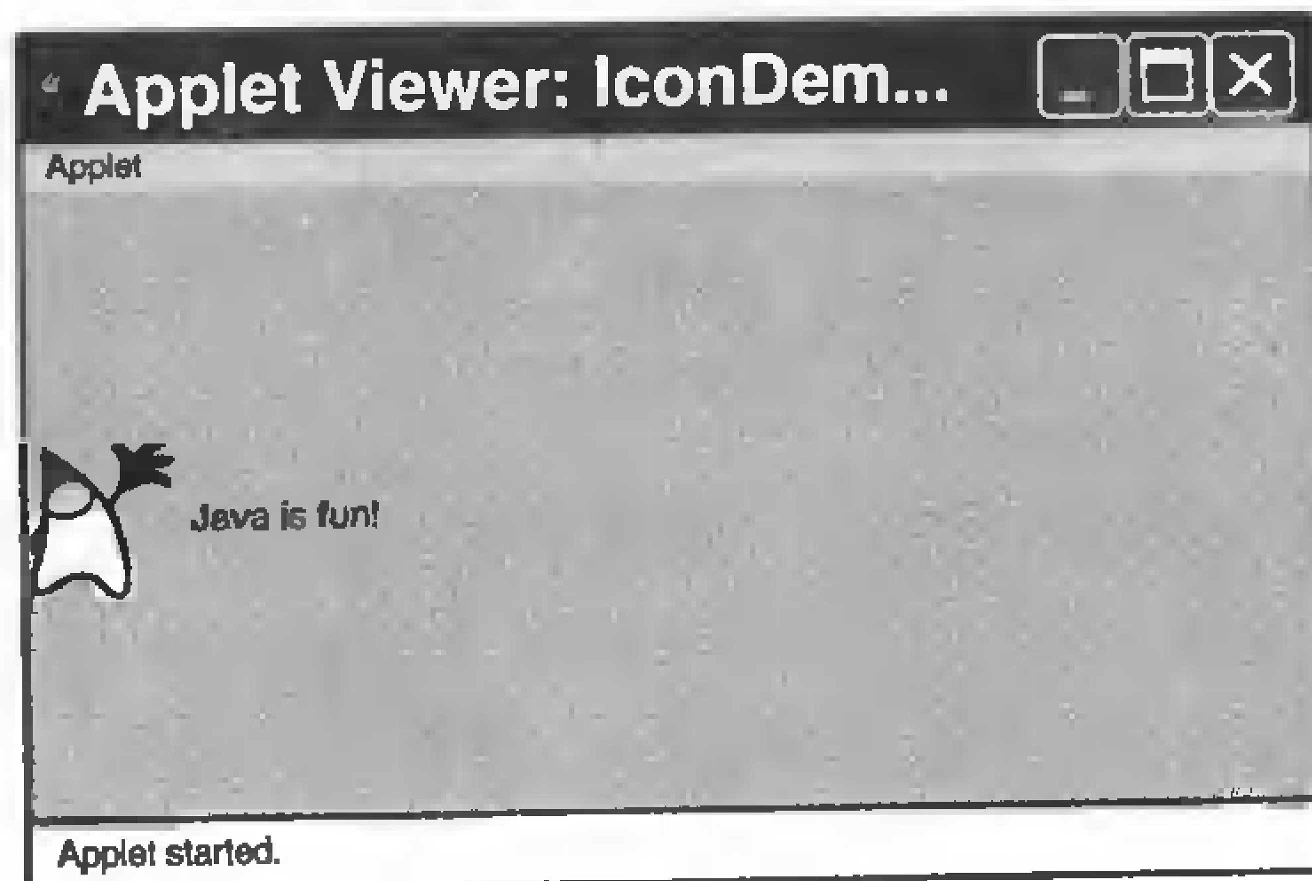


图5-30 带有图标图片的applet

① Java、Duke及所有基于Java的商标和徽标都是Sun公司在美国或其他国家的商标或注册商标（Duke就是在挥手的那个图形。）

标签上可以只有一个图标而没有字符串。例如，如果将图5-30中的代码改成下列形式，字符串"Java is fun!"就不会出现在applet中了：

```
JLabel niceLabel = new JLabel();    //No argument, so no string.
ImageIcon dukeIcon = new ImageIcon("duke_waving.gif");
niceLabel.setIcon(dukeIcon);
```

还可以用方法setIcon向按钮上添加一个图标。例如，将图5-28中的下列代码：

```
JButton sunnyButton = new JButton("Sunny");
contentPane.add(sunnyButton);
sunnyButton.addActionListener(this);
```

改成下列形式：

```
JButton sunnyButton = new JButton("Sunny");
ImageIcon smileyFaceIcon = new ImageIcon("smiley.gif");
sunnyButton.setIcon(smileyFaceIcon);
contentPane.add(sunnyButton);
sunnyButton.addActionListener(this);
```

那么，带有文本"Sunny"的按钮上也会显示出文件smiley.gif中的图片，这样applet会如图5-31所示。可以在Web上找到的、本书的源代码中包含的ButtonIcon-Demo.java中有完整的applet代码。

得到的applet

这个applet的代码在Web上源代码中的文件ButtonIconDemo.java中。



图5-31　带有图标的按钮

自测题

45. 前面介绍了如何修改图5-28中的applet代码，才能使标有“Sunny”字样按钮像图5-31所显示的那样，既包含图片smiley.gif，也包含文本。进一步修改代码，使标有“Cloudy”字样的按钮包含图片nasty.gif和文本。（文件nasty.gif包含在Web上提供的源代码中。）

5.8.5　改变可见性

每个标签和几乎所有其他组件（比如稍后会介绍的按钮和其他组件）中都有一个名为setVisible的方法，可以用来将标签这样的组件从可见改成不可见的，或者从不可见改成可

见的。

如果标签或其他组件用实参true调用了方法setVisible，就会使这个组件可见；如果标签或其他组件用实参false调用了方法setVisible，就会使这个组件不可见。如果没有调用方法setVisible，这个组件将是可见的。

下面这个编程示例说明了方法setVisible的用法。

编程示例：改变可见性的例子

图5-32包含了一个applet，这个applet中带有一个开始不可见的标签，但点击applet中的按钮时，这个标签就会变成可见的。这个标签名为response，是方法init中的下列setVisible调用使其成为不可见的：

```
response.setVisible(false);
```

点击按钮时，会调用方法actionPerformed，这个方法中包含了下列代码行：

```
response.setVisible(true);
```

这就会使标有“Thanks.That felt good!”字样的标签成为可见的。

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

/**
 Simple demonstration of changing visibility in an Applet.
*/
public class VisibilityDemo extends JApplet implements ActionListener
{
    private JLabel response;
    private Container contentPane;

    public void init()
    {
        contentPane = getContentPane();
        contentPane.setBackground(Color.WHITE);

        //Program button:
        JButton aButton = new JButton("Push me!");
        aButton.addActionListener(this);

        //Program label:
        response = new JLabel("Thanks. That felt good!");
        ImageIcon smileyFaceIcon = new ImageIcon("smiley.gif");
        response.setIcon(smileyFaceIcon);
        response.setVisible(false);

        //Add button:
        contentPane.setLayout(new FlowLayout());
        contentPane.add(aButton);

        //Add label
```

标签response和变量contentPane都是实例变量，这样就可以将其用于方法init和actionPerformed了。

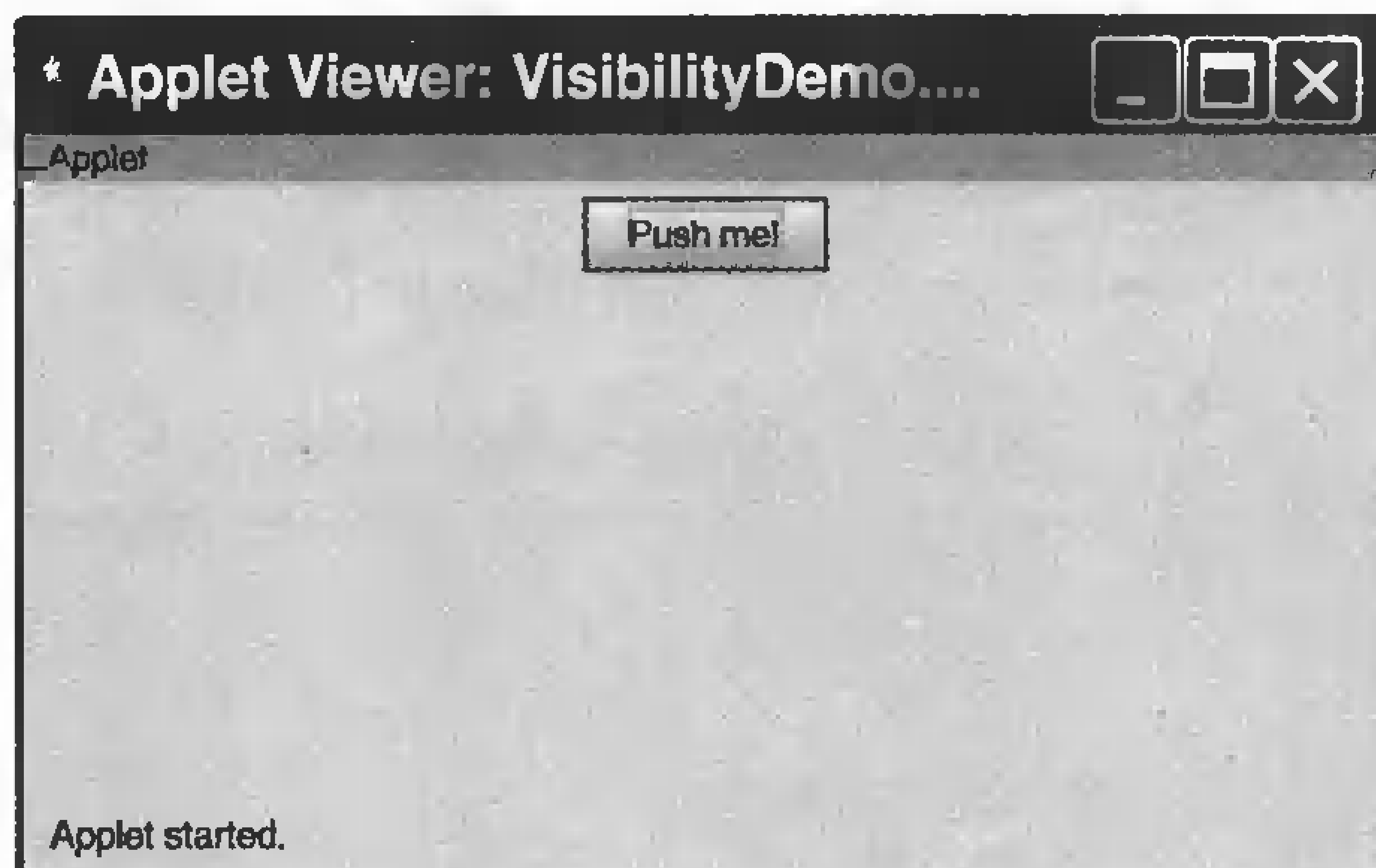
图5-32 可以改变可见性的标签

```
        contentPane.add(response);

    }

    public void actionPerformed(ActionEvent e)
    {
        contentPane.setBackground(Color.PINK);
        response.setVisible(true);
    }
}
```

第一次运行时得到的applet



点击按钮之后得到的applet

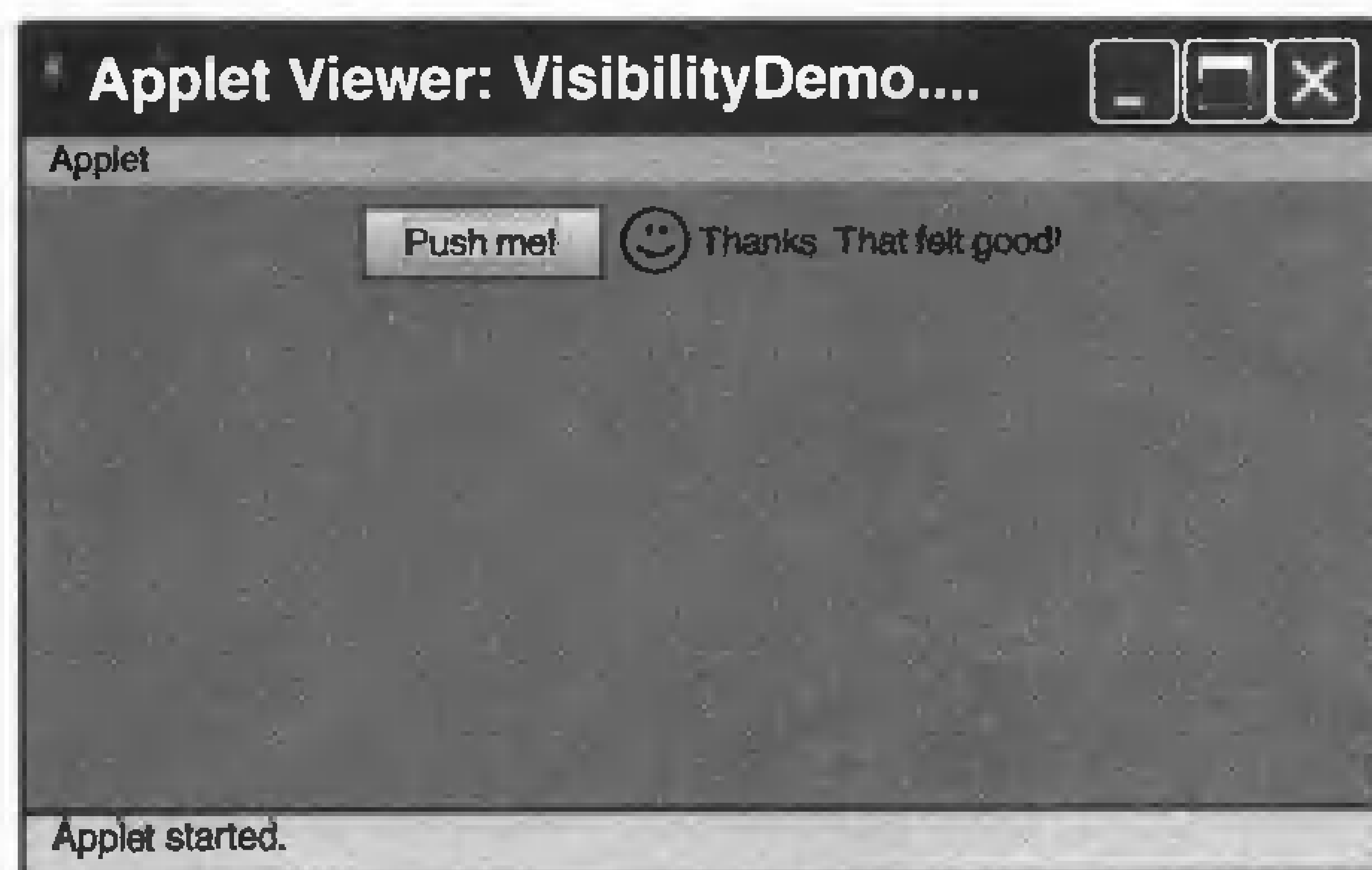


图5-32 （续）

注意，标签response和变量contentPane都是applet中的私有实例变量。这样在方法init和actionPerformed中就都可以使用它们了。

通过init方法中的getContentPane调用，将变量contentPane初始化为applet的内容面板。对变量contentPane的这次初始化是init方法的行为与构造器类似的一个例子。

快速参考：setVisible方法

标签、按钮和稍后会介绍的其他applet组件都有一个setVisible方法。setVisible方法有一个boolean类型的实参。如果aLabel是这些组件之一，

```
aLabel.setVisible(true);
```

会使aLabel成为可见的。调用

```
aLabel.setVisible(false);
```

会将aLabel隐藏起来，即会使aLabel成为不可见的。

当一个组件（比如一个按钮）不可见时，就不能点击它以得到一个动作。“不可见的”组件不仅仅是看不见的，而是它根本就不在那儿。

▲ 易犯错误：“不可见”的含义

与setVisible一起使用的术语“不可见”的含义需要澄清。假设aButton是一个按钮，那么，在执行

```
aButton.setVisible(false);
```

之后，按钮就是“不可见”的，但它在这里的含义比传统意义上“不可见”的含义要更丰富一些。实际上它意味着按钮不在那儿。有些程序员会错误地认为尽管看不见这种“不可见”的按钮，但它们是在那儿的；因此，如果他们点击了屏幕上原来放置按钮的位置，还希望得到与点击按钮相同的效果。那么他们错了，因为无法点击一个不可见的按钮，除非先使其可见。△

自测题

46. 在图5-32中，从没用过方法actionPerformed中的参数e。这样可以吗？

47. 为什么图5-32的方法actionPerformed中没有分支语句？

5.8.6　后续内容

接下来不一定要学习第6章的内容，细节请参看第6章的预备知识。但是，即便没有阅读第6章中的任何内容，你仍然可以学习6.6.1节。如果你在学习图形编程补充部分，那么，即使不学习第6章中的其他内容，也应该学习6.6.1节。

小　结

- 方法的定义中可以包含对同一个类中其他方法的调用。
- 如果一个方法定义被标记为static，就可以用类名，而不是对象名来调用这个方法。（也可以用对象名来调用它。）
- 静态变量是一个用关键字static声明的变量。对每个静态变量名来说，只有唯一一个同名变量为类的所有对象所共享。
- 自顶向下的设计方法通过把方法要完成的任务分解成子任务，来帮助编写方法定义。
- 应该将每个方法都放在一个此方法是唯一未测方法的程序中进行测试。
- 每个基本类型都有一个包装类，作为基本类型的类版本来使用。
- 从Java 5.0开始，需要的时候，Java都会自动地执行从基本类型值到其相应包装类对象的类型转换（及相反的转换）。
- 在同一个类中，一个方法名可以有两个不同的定义，只要这两个定义的参数数量不同，或者某些参数的类型不同就行了。这称为方法名的重载。

■ 构造器是在用new创建类对象时调用的一个类方法。构造器必须拥有与类相同的名字。
■ 没有参数的构造器被称为默认构造器。类定义中通常会包含一个默认构造器。
■ 你可以构建一个你常用的类定义的包。然后，就可以在任意程序中使用它们，而不需要将它们移到与程序相同的目录（文件夹）。
■ 可以向applet中添加一些按钮和图标。[①]
■ 在事件驱动编程中，某些动作会激发事件，比如点击按钮的动作。那么，执行动作的侦听器会收到这些事件，这个过程的细节取决于被激发的事件。
■ 可以用方法setVisible使一些组件（比如标签）可见或不可见。

✓ 自测题答案

1. 是的，如图5-1所示。
2. 没有任何影响。在这种上下文中，下面两行代码是等效的。
```
seekAdvice();
```
和
```
this.seekAdvice();
```
3. 这会产生一条“Null指针异常”错误消息。变量s1和s2没有命名任何对象。应该将代码行
```
Species s1 = null;
Species s2 = null;
```
改成
```
Species s1 = new Species();
Species s2 = new Species();
```
4. 这是合法的，但更常见的方法是：
```
double areaOfCircle = CircleFirstTry.area(2.5);
```
5. 是的。可以在一个类中包含所有这些方法。
6. 除非你创建了一个类对象，并用那个对象作为非静态方法的调用对象，否则就无法在静态方法定义中调用非静态方法。
7. 是的，你可以在非静态方法定义中调用静态方法。对此没有特殊要求。
8. 不行，因为调用静态方法时可以不使用调用对象，而如果没有调用对象，就不会有实例变量，所以，不能在静态方法定义中引用实例变量。
9. 不，下列代码是非法的，因为setDiameter不是一个静态方法：
```
PlayCircle.setDiameter(newDiameter);
```
10. *实例变量*（instance variable）是在类定义中不使用关键字static声明的。*静态变量*（static variable）是在类定义中使用关键字static声明的。类的每个对象都有它自己的*实例变量*。类的每个*静态变量*都只有一个，类的所有对象都共享那个静态变量。
11. 是的，可以在*静态方法*（static method）的定义中（通过名字，而不使用任何类名和点）引用*静态变量*。图5-8所示方法justADemoMethod中的静态变量numberOfInvocations就是一个例子。
    不，不能在*静态方法*（static method）的定义中（通过名字，不使用任何类名和点）引用*实例变量*。这是因为可以通过类名而不是调用对象来使用静态方法，而没有调用对象，就不会有实例变量。
12. 是的，可以在*非静态方法*（nonstatic method）的定义中（通过名字，不使用任何类名和点）引用*静态变量*。是的，可以在*非静态方法*（static method）的定义中（通过名字，不使用任何类名和点）引用*实例变量*。注意，这意味着你可以在非静态方法中使用这两种类型的变量。

① 5.8节中介绍的内容。

13. `2, 3, 2.0,`
   `2.0, 3.0, 3.0.`
   注意，前两个值是long类型的，而后面4个值是double类型的。
14. `approxSpeed = (int)Math.round(speed);`
   这是因为Math.round会返回一个long类型的值。
15. `longSpeed = Math.round(speed);`
   换句话说，不需要做什么特别的事情。
16. long。因为一个实参是long类型的，结果就是long类型的。
17. 都是合法的。
18. `Double.toString(x)`
19. `Integer.parseInt(s)`
20. 如果字符串中可能包含前导或结尾空格，就应该使用Integer.parseInt (s.trim())。实际上，使用这种格式通常是最安全的。你永远都不会知道什么时候会出现前导或结尾空格。
21.
```
System.out.println("Largest double is " + Double.MAX_VALUE);
System.out.println("Smallest double is " + Double.MIN_VALUE);
```
22. Web上提供的源代码中包含了文件OutputFormat.java中的OutputFormat类。代码如下：
```
public class OutputFormat
{
    /**
     Writes out number with digitsAfterPoint digits after
     the decimal point. Rounds any extra digits.
     Does not advance to the next line after output.
    */
    public static void write(double number,
                                 int digitsAfterPoint)
    {
        if (number >= 0)
            writePositive(number, digitsAfterPoint);
        else
        {
            double positiveNumber = -number;
            System.out.print('-');
            writePositive(positiveNumber, digitsAfterPoint);
        }
    }
    //Precondition: number >= 0
    //Writes out number with digitsAfterPoint digits after the
    //decimal point. Rounds any extra digits.
    private static void writePositive(double number,
                                         int digitsAfterPoint)
    {
        int mover = (int)(Math.pow(10, digitsAfterPoint));
                  //1 followed by digitsAfterPoint zeros
        int allWhole; //number with the decimal point
                      //moved digitsAfterPoint places
        allWhole = (int)(Math.round(number*mover));
        int beforePoint = allWhole/mover;
        int afterPoint = allWhole%mover;
        System.out.print(beforePoint);
```

```
        System.out.print('.');
        writeFraction(afterPoint, digitsAfterPoint);
    }
    //Outputs the integer afterPoint with enough zeros
    //in front to make it digitsAfterPoint digits long.
    private static void writeFraction(int afterPoint,int digitsAfterPoint)
    {
        int n = 1;
        while (n < digitsAfterPoint)
        {
            if (afterPoint < Math.pow(10, n))
                System.out.print('0');
            n = n + 1;
        }
        System.out.print(afterPoint);
    }
    /**
     Writes out number with digitsAfterPoint digits after
     the decimal point. Rounds any extra digits.
     Advances to the next line after output.
    */
    public static void writeln(double number,int digitsAfterPoint)
    {
       write(number, digitsAfterPoint);
       System.out.println( );
    }
}
```

23. 是的，可以在OutputFormat类中使用名字print和println，而不是write和writeln。这样不会与System.out.println产生名字冲突，因为在调用OutputFormat中的方法时，在点前面指定了类名。（用对象而不是类名来调用方法时，由于Java知道对象的类型，所以Java还是会知道类名的。）但是，OutputFormat中方法的行为与System.out.println方法有些不同，因此，使用不同的名字可能会更清晰一些。
24. 使用long类型的变量，而不是int类型的变量。注意，因为在方法writePositive中，方法Math.round返回的是一个long类型的值，所以，这样做甚至会节省一次强制类型转换。例如：

```
int allCents = (int)(Math.round(amount*100));
```

会变成

```
long allCents = (Math.round(amount*100));
```

变量dollars和cents也应该改成long类型的。
25. 可以，Java会将7改成7.0，以使这些类型与一个set定义的头部相匹配。
26. 不可以，不能在返回值的基础上重载方法名。
27. 可以，它们的参数类型不同，因此这是对方法名convertValue的合法重载。（注意，它们返回不同类型的值对能否使用这两个定义没有影响。只有参数类型不同才能使重载合法。）
28. 可以，因为没有其他名为set的方法具有相同数量和类型的参数，所以它是合法的。定义如下所示：

```
public void set(String newName)
{
    name = newName;
    population = 0;
```

```
    growthRate = 0;
}
```

29. 可以，因为没有其他名为set的方法具有相同数量和类型的参数，所以它是合法的。定义如下所示：

```
public void set(String newName)
{
    name = newName;
}
```

30. 不行，如果你将这两个新的set方法都添加进去，类中会有两个具有相同参数数量及类型的、名为set的方法定义。
31. 只要从定义中删除所有的this.就行了。
32. 只要添加对trim方法的调用就行了。重新编写的代码如下：

```
public void set(String amountString)
{
    String dollarsString;
    String centsString;
    amountString = amountString.trim();
    <方法定义的其余部分与图5-18相同>
```

33. 如果类被命名为Student，那么这个类的每个构造器也都必须被命名为Student。
34. 不用为构造器指定返回类型，甚至连void也不用。
35. 默认构造器是不带参数的构造器。
36. 不是的。细节如下所示：如果你没有给出一个类的构造器定义，Java会自动地提供一个默认构造器。如果提供了一个或多个任意类型的构造器，那么，除了你定义的构造器之外，Java就不会再提供任何构造器了。因此，如果定义了一个或多个构造器，而其中没有一个是默认构造器，这个类就没有默认构造器了。
37.

```
public String getPetName()
{
    return pet.getName();
}
public int getPetAge()
{
    return pet.getAge();
}
public double getPetWeight()
{
    return pet.getWeight();
}
```

38. `import mypackages.library1.*;`
39. 必须将下列行作为文件的第一条指令行：

```
package mypackages.library1;
```

40. 包名必须是包含包中类的目录的路径名，但在包名中用点取代了\或/（不管你的操作系统使用了其中哪一个）。在命名包的时候，使用的是从类路径（环境）变量设置中命名的任意一个目录开始的相对路径名。
41. 完成这项任务的方式与你所用的操作系统及个人偏好没什么关系。下面是你需要做的工作：选择一个包名，并将下列代码插入文件Species.java的起始处：

    package *Whatever_Package_Name_You_Choose*;

    然后编译修改过的文件Species.java，并将文件Species.java和Species.class都移到与

*Whatever_Package_Name_You_Choose*相对应的目录。

42. 可以将内容面板命名为你所期望的（除关键字之外的）任何名字。因此下列代码是完全合法的：

```
Container insideOfApplet = getContentPane();
```

（当然，如果进行了这样的修改，就应该用insideOfApplet替换在方法actionPerformed中出现的所有其他contentPane。）

43. 事件被“发送”到一个侦听器对象时，实际上是指：用事件对象作为实参调用了侦听器对象中的某些方法。这种调用是自动发生的。applet类定义通常不会包含对这个方法的调用。

44. 对程序不会有什么影响。e是方法actionPerformed的参数，可以用任何（非关键字）标识符作为参数。

45. 将代码

```
JButton cloudyButton = new JButton("Cloudy");
contentPane.add(cloudyButton);
cloudyButton.addActionListener(this);
```

改成下列代码：

```
JButton cloudyButton = new JButton("Cloudy");
ImageIcon nastyFaceIcon = new ImageIcon("nasty.gif");
cloudyButton.setIcon(nastyFaceIcon);
contentPane.add(cloudyButton);
cloudyButton.addActionListener(this);
```

完整的applet代码在Web上本书源代码中包含的文件ButtonIconDemo2.java中。

46. 是的，这样可以。参数e必须包含在方法头部，但不一定要在方法主体中使用。

47. 侦听器（即applet自己）只侦听了一个按钮，所以它不需要确定是哪个按钮被点击了。只有一个按钮，所以只有一个动作。

● 编程项目

1. 为输出double类型的值定义一个实用工具类，将这个类称为DoubleOut。这个类要包含图5-14中Dollars类的所有方法；自测题22中OutputFormat类的所有方法；以及一个名为scienceWrite的方法，这个方法会以e记数法输出一个double类型的值，比如2.13e-12。（这种e记数法也被称为科学表示法，这也解释了方法名的含义。）以e记数法输出时，总是只在小数点前面显示一位非零的数字（除非数字正好等于零）。方法scienceWrite的输出不会输出到下一行。还要添加一个名为scienceWriteln的方法，除了接下来的输出会输出到下一行之外，这个方法与scienceWrite一样。除了最后两个方法定义之外，其他方法都可以简单地从正文中复制（或者更简单地从Web上本书的源代码中复制）。注意，要重载write和writeln。编写一个驱动程序来测试方法scienceWriteln。该驱动程序应该为方法scienceWrite使用一个存根。（注意，这就意味着你甚至可以在编写scienceWrite之前编写并测试scienceWriteln。）然后，为方法scienceWrite编写一个驱动程序。最后，编写一个超级驱动程序之类的程序，将一个double值作为输入，然后用两个writeln方法和scienceWrite方法将其输出。当需要指定小数点后面数字的个数时，使用数字5。这个超级驱动程序应该允许用户用其他的double类型数字进行这种测试，直到用户准备结束这个程序为止。

2. 修改图4-19中Species类的定义，从中删除方法set，并将下列方法添加进去。

   (1) 5个构造器：每个实例变量一个构造器，一个用于3个实例变量的、具有3个参数的构造器，以及一个默认构造器。

   (2) 4个可以对值进行重新设置的、名为set的方法：一个与图4-19中的set方法相同，其他3个分别重

新设置3个实例变量。

确保每个构造器都设置了所有的实例变量。然后编写一个测试程序来测试你添加的所有方法。最后，重新完成（或者第一次完成）第4章第1个编程项目。在定义Species类的新对象时，一定要使用除默认构造器之外的其他一些构造器。

3. 重新完成（或第一次完成）第4章的编程项目4。这一次要确保你的类定义中包含了合适的构造器和重置方法。
4. 重新完成（或第一次完成）第4章的编程项目4。这一次，除了重置和"测试"（同名、同龄、较老或较年轻的）方法之外，还要添加下列4个构造器：每个实例变量一个构造器，一个用于两个实例变量的、具有两个参数的构造器，以及一个默认构造器。确保每个构造器都设置了所有的实例变量。编写一个测试程序来测试每个方法，包括这4个构造器，并至少用一个为真和一个为假的情况来测试每个测试方法。
5. 编写一个新的类TruncatedDollars，除了通过截尾而不是四舍五入来获得小数点后的两位数字之外，这个类与图5-14中的Dollars类相同。（截尾时，前两个数字之后的所有数字都被删除了，因此1.229会变成1.22，而不是1.23。）用这个类来重新实现（或第一次实现）第3章的编程项目6。
6. 编写一个程序读入5个宠物的数据（使用图5-20中的PetRecord类），并显示下列数据：最小的宠物的名字，最大的宠物的名字，年龄最老的宠物的名字，年龄最小的宠物的名字，5个宠物的平均重量，以及5个宠物的平均年龄。
7. 编写一个带有两个参数的Temperature类：一个温度值（浮点数）和一个用来表示度量的字符（表示摄氏度的'C'，或者表示华氏度的'F'）。这个类应该有4个构造器：每个实例变量一个构造器（如果没有指定任何值就假定是零度，如果没有指定度量就假定是摄氏度），一个用于两个实例变量的、具有两个参数的构造器，以及一个默认构造器（设置为零摄氏度）。类中包括以下内容。
   （1）两个用来返回温度的访问方法，一个用来返回摄氏度，另一个用来返回华氏度。用第3章编程项目2中的公式来编写这两个方法，并将其四舍五入到最近的1/10度上去；
   （2）3个重置方法，一个用来设置温度值，一个用来设置度量（'F'或'C'），还有一个用来对这两者进行设置。
   （3）3个比较方法，一个用来测试两个温度是否相等，一个用来测试一个温度是否高于另一个，一个用来测试一个温度是否低于另一个。

   然后，编写一个用来测试所有方法的驱动程序。一定要用到每个构造器，至少用一个为真和一个为假的情况来测试每个比较方法，并至少对下列温度值的相等性进行测量：0.0°C＝32.0°F，－40.0°C＝－40.0°F，以及100.0°C＝212.0°F。
8. （这个项目需要使用5.8节中的内容）修改图5-32中的applet，使点击按钮之后，按钮消失。（与图5-32一样，带有图标的标签仍然是可见的。）（提示：不需要进行太大的修改。）
9. （这个项目需要使用5.8节中的内容）为一个带有3个按钮的applet编写代码，这3个按钮被标记为Red、White和Blue。点击一个按钮时，applet的背景会变成按钮上所标明的颜色。

第6章

数　　组

他们整齐地列队立正，全部是同样的制服，只是各有自己的价值观。

——瓦恩·皮斯，美国流行歌曲词曲作家，《中尉的队列》

数组（array）是用来存储批量（可能相当大）数据的一种特殊对象。数组在以下两方面和其他对象有所不同：

（1）数组中存储的数据必须是同样的类型。例如，可以用数组存储一系列`double`类型的数值，记录以厘米表示的降雨量数值；还可以用数组存储一系列类型为`Species`的对象，记录各种濒危物种。

（2）数组对象只有少数预定义好的方法。因为在发明类之前，程序员们就已经使用这些方法好多年了，所以使用数组的这些预定义方法时，采用了特殊的记号。大部分人甚至不称它们为方法。

本章将介绍数组，并展示如何在Java中使用数组。

目标

- 了解数组以及如何在简单的Java程序中使用数组。
- 学习如何采用数组作为参数，以及如何定义方法以返回数组。
- 学习如何正确地把数组作为一个实例变量用在类中。
- 了解在数组中排序的专题内容。
- 熟悉多维数组。
- 选读，在applet中插入文本域和文本区。
- 选读，在applet中画任意的多边形。

预备知识

阅读6.1节仅需要先读第1章～第3章。应当熟练掌握前面的5章才能学习本章的其他部分(6.2节～6.5节)。

在本书中，这是第一个让读者相当自由地选择下一步要阅读内容的地方。如果喜欢，可以先读本书后面的部分，以后再回来阅读本章。在阅读本章之前，读者可以根据自己的喜好阅读第7章和本书姊妹篇《Java程序设计与问题解决：高级篇（第4版)》。

如果学习每章的“图形编程补充”，则不管是否读过本章以前的章节，都可以开始阅读6.6.1节的，也建议如此。

6.1 数组基础

在这样的索引下，虽然只是指示随后情景的一些小点点，
可是因小见大，可以看出将来自由发展成怎样的局势。
——威廉·莎士比亚，《特洛埃勒斯与克蕾雪达》（中文译文参考梁实秋译本）

假设需要计算1周7天的平均温度，则可以编码如下：

```
int count;
double next, sum, average;
Scanner keyboard = new Scanner(System.in);
System.out.println("Enter 7 temperatures:");
sum = 0;
for (count = 0; count < 7; count++)
{
    next = keyboard.nextDouble();
    sum = sum + next;
}
average = sum/7;
```

若只是想了解平均温度，这段代码可以顺利工作。但是假如还需要知道哪些天温度高于平均值，哪些天温度低于平均值，问题就来了。为了计算平均温度，必须读入7天的温度，而且为了把每天的温度与平均值相比较，还要先求出这个平均温度。这样，为了进行比较，就必须记住这7天的温度。如何才能做到呢？很容易想到的方法是用7个double类型的变量。这样做略显繁琐，因为声明7个变量显然太多了，而且在其他情况下，或许问题更糟糕。想象一下如果不是对1周而是要对1年的气温值进行计算，声明365个变量是多么荒唐。数组提供了一种优雅的方法来声明一组相关的变量。**数组**就像是一系列变量，但它优美而简洁地解决了变量命名的问题。

6.1.1 创建与访问数组

在Java中，数组是一种特殊的对象。但把它看成是一组同样类型的变量的集合往往更好。例如，充当7个double类型变量集合的数组可以创建如下：

```
double[] temperature = new double[7];
```

这就和下面声明7个double类型的变量类似：

```
temperature[0], temperature[1], temperature[2], temperature[3],
temperature[4], temperature[5], temperature[6]
```

注意，*编号是从0开始的，不是1*。这7个变量都可以和其他任何double类型的变量一样使用。例如，下面的代码在Java中都是合法的：

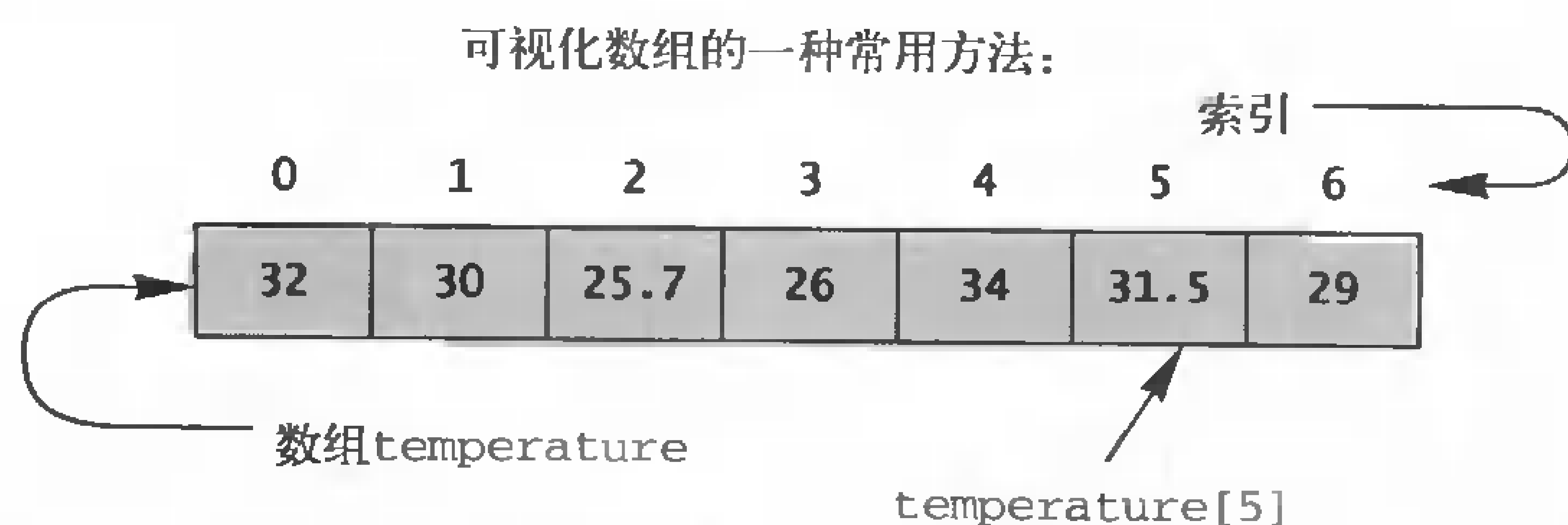
```
temperature[3] = 32;
temperature[6] = temperature[3] + 5;
System.out.println(temperature[6]);
```

不过这7个变量不仅仅是7个普通的double类型的变量，方括号中的数字允许通过计算来求出其中一个变量的名字。用任何得出0和6之间的整数的表达式都可以替换方括号内的整数常量，下面的代码都是合法的：

```
Scanner keyboard = new Scanner(System.in);
System.out.println("Enter day number (0-6):");
int index = keyboard.nextInt();
System.out.println("Enter temperature for day " + index);
temperature[index] = keyboard.nextDouble();
```

像`temperature[0]`、`temperature[1]`这样的带有内置整数表达式的方括号的变量，有多种名称。本书称为**索引变量**（indexed variable）或**元素**(element)。有些人称其为**下标变量**(subscripted variable)。方括号内的整数表达式称为**索引**（index）或者**下标**（subscript）。当我们把这些被索引的变量组合在一起作为一个整体看待时，就称为数组，因此可以将这个数组叫做`temperature`(不再使用方括号)。

图6-1所示的程序例子中，用样例数组`temperature`作为7个被索引的`double`类型的变量。



屏幕对话示例

```
Enter 7 temperatures:
32
30
25.7
26
34
31.5
29
The average temperature is 29.7428
The temperatures are
32.0 above average
30.0 above average
25.7 below average
26.0 below average
34.0 above average
31.5 above average
29.0 below average
Have a nice week.
```

图6-1　在程序中使用的数组

注意，程序可以用一个变量作为索引来计算索引变量名，就像下面在`for`循环中做的一样：

```
for (index = 0; index < 7; index++)
{
    temperature[index] = keyboard.nextDouble();
    sum = sum + temperature[index];
}
```

6.1.2 数组的细节

创建数组就像创建一个类类型的对象一样，但是使用的记号稍有不同。创建以*Base_Type*作为元素类型的数组，语法如下：

Base_Type [ ] *Array_Name* = new *Base_Type* [*Length*];

例如，下面的代码创建了一个名叫pressure的数组，相当于100个int类型的变量。

```
int[] pressure = new int[100];
```

```
import java.util.*;
public class ArrayOfTemperatures
{
    /**
     Reads in 7 temperatures and shows which are above and
     which are below the average of the 7 temperatures.
    */
    public static void main(String[] args)
    {
        double[] temperature = new double[7];

        int index;
        double sum, average;
        Scanner keyboard = new Scanner(System.in);
        System.out.println("Enter 7 temperatures:");
        sum = 0;
        for (index = 0; index < 7; index++)
        {
            temperature[index] = keyboard.nextDouble();
            sum = sum + temperature[index];
        }
        average = sum/7;

        System.out.println("The average temperature is "+ average);
        System.out.println("The temperatures are");
        for (index=0; index < 7; index++)
        {
            if (temperature[index] < average)
                System.out.println (temperature[index] + " below average.");
            else if (temperature[index] > average)
                System.out.println (temperature[index] + " above average.");
            else //temperature[index] == average
                System.out.println (temperature[index] + " the average.");
        }
        System.out.println("Have a nice week.");
```

```
    }
}
```

另一种方法是把创建分为两步：

```
int[] pressure;
pressure = new int[100];
```

元素的类型，即在本例中的int，称为数组的**基类型**（base type）。数组中元素的数目称为数组的**长度**（length）或**大小**（size）。示例数组pressure的长度是100，即它有索引变量从pressure[0]直到pressure[99]。注意，索引是从0开始的，所以像pressure这样长度是100的数组，并不包含索引变量pressure[100]。

数组的基类型可以是任何类型，尤其可以是类。下面的代码创建了一个名为entry的数组，相当于一组3个Species类型的变量：entry[0]、entry[1]及entry[2]（这里Species是某个类）：

```
Species[] entry = new Species[3];
```

快速参考：声明和创建数组

声明一个数组名称并创建数组，同创建并命名一个类的对象几乎是一样的，只是在语法上有少许不同。

语法：

Base_Type[] *Array_Name* = new *Base_Type*[*Length*];

举例：（Species是一个类）

```
char[] symbol = new char[80];
double[] reading = new double[100];
Species[] specimen = new Species[80];
```

数组名外侧加方括号有3种用法，不能混淆。首先，方括号可以用于创建一个类型名称，比如下面的int[]：

```
int[] pressure;
```

其次，方括号可以与一个整数值一起，作为Java语言中创建一个新数组对象的语法的一部分，例如：

```
pressure = new int[100];
```

方括号的第三种用法，是命名数组中的索引变量，如pressure[0]或pressure[3]，参见下面两行代码：

```
pressure[3] = keyboard.nextInt();
System.out.println("You entered " + pressure[3]);
```

前面已经提到，方括号中的整数可以是任何能求出整数值的表达式。例如，可以像下面的代码那样，从键盘读入数组的长度：

```
System.out.println("How many temperatures will there be?");
int size = keyboard.nextInt();
double[] temperature = new double[size];
```

在其他情形中，方括号中的整数也是同样的，可以用能求出合适整数值的任何表达式，例如：

```
int point = 2;
```

```
temperature[point + 3] = 32;
System.out.println(
        "Temperature 5 is " + temperature[point + 3]);
```

注意，在上面的代码中，temperature [point + 3]和temperature[5] 是同一个索引变量，因为point + 3得出的结果就是5。

图6-2中显示了大多数关于数组的常用术语的意义。注意元素（element）有两种意思。一个索引变量叫作元素，而索引变量的值有时也叫作元素。

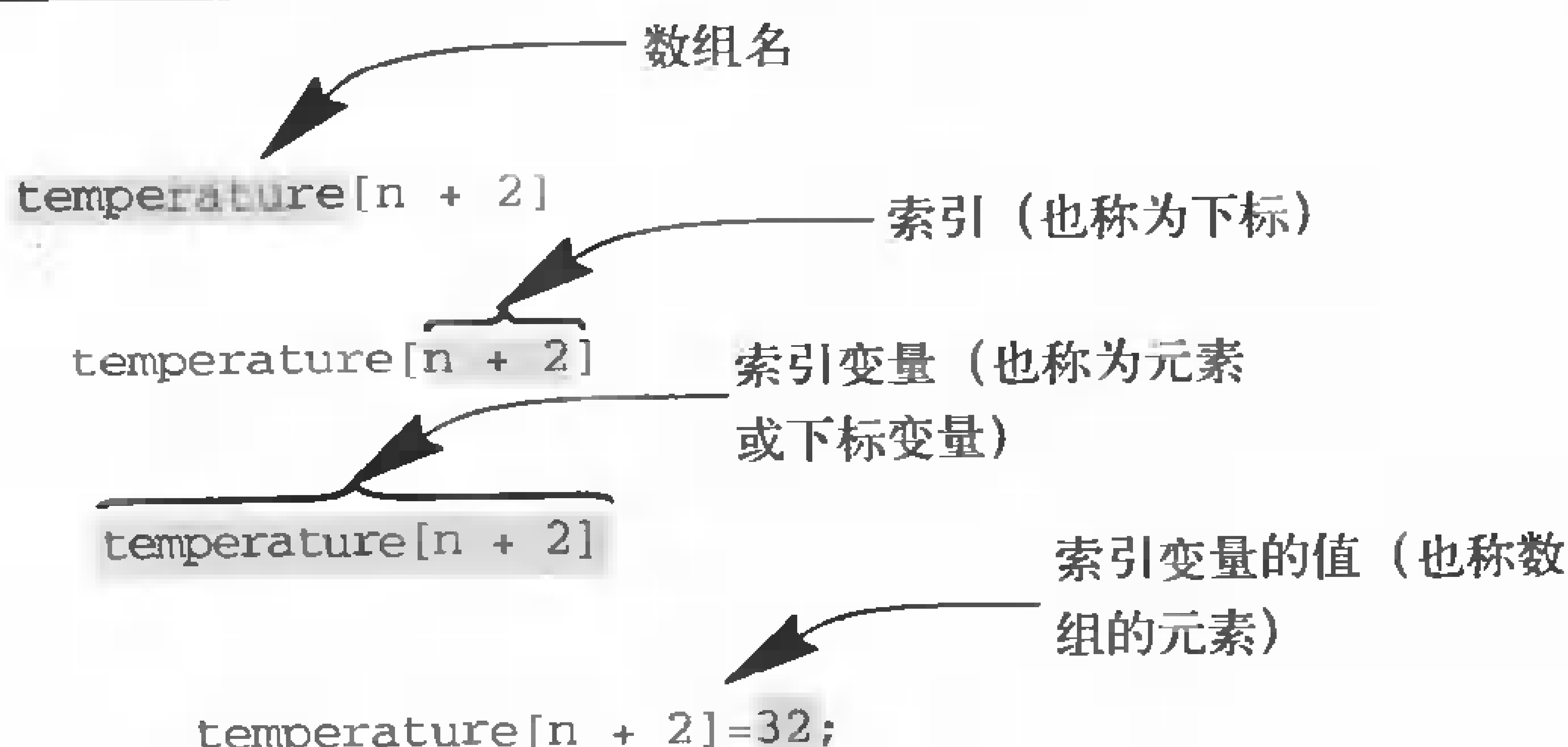


图6-2 数组的各种术语

● 编程提示：数组名使用单数

假设要在数组中放置一些Species类的对象，你可能想这样写代码：

```
Species[] entries = new Species[20]; //Valid but not nice.
```

使用名词的复数形式，比如entries，看起来挺合理的，因为数组中包含不止一个元素。然而，程序员往往会发现若使用名词的单数形式，程序读起来更好，比如：

```
Species[] entry = new Species[20]; //Nicer.
```

使用单数形式更好，是由于在一类计算中，数组名仅用来指其中的一个元素。表达式entry[2]是数组的单个元素，例如：

```
System.out.println ("The entry is " + entry[2]);
```

数组名使用单数形式也不是绝对的，有时候复数形式更合理。例如，若数组的索引变量包含的是第n个雇员工作时间的小时数，则复数形式hours[n]更合适。用单数形式还是复数形式更好，要看把索引变量放在Java代码的上下文中读起来是什么样来检验。

6.1.3 实例变量length

数组是一种对象，与其他对象一样，可能有实例变量。实际上，每个数组仅有一个公有的实例变量，名为length，它的值等于数组的长度。例如，若像下面这样创建数组：

```
Species[] entry = new Species[20];
```

则entry.length的值为20。

使用实例变量length来代替常量，比如这个20，可以让程序更加清晰。因为后者的意思并不总是很明确，而entry.length就明白多了。在图6-3中，我们用实例变量length重写了图6-1中的程序。

```
import java.util.*;

public class ArrayofTemperatureS2
{
    /**
     Reads in 7 temperatures and shows which are above and
     which are below the average of the 7 temperatures.
    */
    public static void main(String[] args)
    {
        double[] temperature = new double[7];
        int index;
        double sum, average;
        Scanner keyboard = new Scanner(System.in);
        System.out.println("Enter " + temperature.length
                                    + "temperatures:");
        sum = 0;
        for (index = 0; index < temperature.length; index++)
        {
            temperature[index] = keyboard.nextDouble();
            sum = sum + temperature[index];
        }
        average = sum/temperature.length;

        System.out.println("The average temperature is "
                                                + average);
        System.out.println("The temperatures are");
        for (index = 0; index < temperature.length; index++)
        {
            if (temperature[index] < average)
                System.out.println(
                          temperature[index] + " below average.");
            else if (temperature[index] > average)
                System.out.println(
                          temperature[index] + " above average.");
            else //temperature[index] == average
                System.out.println(
                          temperature[index] + " the average.");
        }
        System.out.println("Have a nice week.");
    }
}
```

屏幕对话和图6-1相同。

图6-3 实例变量length

实例变量length不能用程序来改写（除非再用new创建一个新的数组），比如下面的代码是不正确的：

```
entry.length = 10;//Illegal!
```

■ Java提示：数组索引从0开始

在Java中，数组的索引总是从0开始。永远也不会从1或者别的非0的数开始。这意味着最后那个元素的索引数值不等于数组的长度，而是长度减去1。

在某些情况下，正常的思维方式是使用另外的编号方案，这就需要调整代码来使数组的索引和直观的编号相对应。例如，可能需要将数组中的数据按照起始编号为1来思考。例如，公司的雇员编号从1开始，则工资单程序可能会像下面的代码这样：

```
int[] hours = new int[100];
Scanner keyboard = new Scanner(System.in);
System.out.println("Enter hours worked for each employee:");
int index;
for (index = 0; index < hours.length; index++);
{
    System.out.println(
                    "Enter hours for employee " + (index + 1));
    hours[index] = keyboard.nextInt();
}
```

这段代码中，雇员编号为1～100，但是它们的工作小时数被保存在元素hours[0]到hours[99]中。

● 编程提示：使用for循环遍历数组

for语句是遍历数组中各个元素的理想方式。例如，下面从图6-3中摘录的for循环就演示了如何遍历数组：

```
for (index = 0; index < temperature.length; index++)
{
    temperature[index] = keyboard.nextDouble();
    sum = sum + temperature[index];
}
```

▲ 易犯错误：数组索引越界

对数组编程时很容易犯的一个错误是索引的表达式求出的值是非法的数组索引。例如下面的代码声明的数组：

```
double[] entry = new double[5];
```

数组entry中的每个索引必须是下列5个整数之一：0，1，2，3，4。例如，程序中使用索引变量entry[n + 2]，则n + 2必须等于这5个整数之一。若索引的表达式求出的值不是从0到数组的长度减1之一，这样的索引就称为**越界**(out of bounds)或者说是**非法的**（invalid)。如果程序中用到的索引表达式越界了，那么在程序（或者类）编译的时候不会出错，但是Java会在运行程序时发现问题。

一种常见的数组索引越界的方式是对数组的处理循环多迭代了一次。这几乎在处理数组的索引的各种循环形式中都会出现，不过一个常见的例子就是在对数组进行循环填充时。例如，下面的代码从键盘读入一系列非负数，并以一个负数作为输入结束的标志：

```
System.out.println("Enter a list of nonnegative integers.");
System.out.println("Place a negative integer at the end.");
int[] a = new int[10];
Scanner keyboard = new Scanner(System.in);
```

```
int number = keyboard.nextInt();
int i = 0;
while (number >= 0)
{
    a[i] = number;
    i++;
    number = keyboard.nextInt();
}
```

如果用户输入的数比数组中能放入的数更多，则这段代码就会产生越界的数组索引。下面的代码是对上面代码的while循环的改进：

```
while ( (i < a.length) && (number >= 0) )
{
    a[i] = number;
    i++;
    number = keyboard.nextInt();
}
if (number >= 0)
{
    System.out.println("Could not read in all the numbers.");
    System.out.println("Only read in " + a.length + " numbers.");
}
```

while循环这样改写后，当数组被充满时，循环就会结束。

注意，在后来的while循环中，对索引i进行了严格检查确保其小于a.length，因为索引是从0开始的，最后元素的索引不是a.length，而是比a.length小1。 △

6.1.4 数组的初始化

数组可以在声明时就初始化，只需把每个索引变量的值按顺序写在花括号中，放在赋值操作符之后，参见下面的代码：

```
double[] reading = {3.3, 15.8, 9.7};
```

数组的长度（大小）设置为恰好能容纳这些值。因此这种声明就初始化的方式等价于下面的语句：

```
double[] reading = new double[3];
reading[0] = 3.3;
reading[1] = 15.8;
reading[2] = 9.7;
```

如果不用代码初始化数组的各个元素，它们会被自动初始化为基类型的默认值。例如，若不用代码初始化一个整数数组，则每个元素都会被初始化为0。不过，用代码显式地进行初始化通常看起来更明白。除了前面说的用花括号进行初始化，还有别的方法，例如：

```
int[] count = new int[100];
int i;
for (i = 0; i < count.length; i++)
    a[i] = 0;
```

自测题

1. 下面的代码会产生什么样的输出结果？

```
int i;
int[] a = new int[10];
for (i = 0; i < a.length; i++)
    a[i] = 2*i;
for (i = 0; i < a.length; i++)
    System.out.print(a[i] + " ");
System.out.println();
```

2. 下面的代码会产生什么样的输出结果?

```
char[] vowels = {'a', 'e', 'i', 'o', 'u'};
int index;
for (index = 0; index < vowels.length; index++)
    System.out.println(vowels[index]);
```

3. 下面的代码会产生什么样的输出结果?

```
double tide[] = {12.2, -7.3, 14.2, 11.3};
System.out.println("Tide 1 is " + tide[1]);
System.out.println("Tide 2 is " + tide[2]);
```

4. 下面的代码对数组b初始化的时候犯了什么错误?

```
int[] b = new int[10];
int i;
for (i = 1; i <= b.length; i++)
    b[i] = 5*i;
```

5. 下面的数组a中，最后一个元素的索引是什么? a.length的值是多少?

```
int[] a = new int[10];
```

6. 写一个完整的Java程序，从键盘读入20个double类型的数，填充到数组中，然后输出数组中的数，并输出每个数与最后一个输入的数的差值。例如，某个元素是2.0，而最后一个输入的数值是5.0，则差值是3.0；若某个元素是7.0，最后一个输入的数值是5.0，则差值就是－2.0。假设用户从键盘输入20个数字，每个一行，不需要在程序中给出详尽的指示。

6.2 在类和方法中使用数组

亲上加亲，但却愈加不够亲切。

——威廉·莎士比亚，《哈姆雷特》(中文译文参考朱生豪译本)

在类中，数组可以被用作实例变量。数组中索引变量和整个数组都可以作为方法的参数。方法可以用数组作为返回值。简而言之，数组在类和方法中的用法像其他对象一样。下面从一个案例分析开始学习，它把数组用在一个类中作为实例变量。

案例分析：销售报表

在这个案例分析中，要写一个程序来产生某公司销售员团队的销售报表。该公司希望能很容易看出哪个或哪些人员的销售成绩最高，并且想看出每个销售员的成绩与平均值的比较关系。

对于每个销售员，必须有一条记录，其中有名字和销售额。因此我们为销售员设计一个类，记录这两项数据，进行输入、输出，以及包含一些访问和设置的辅助方法，这个类在图6-4中列出。该类的定义很平常，不再详述。

```
import java.util.*;
/**
 Class for sales associate records.
*/
public class SalesAssociate
{
    private String name;
    private double sales;

    public SalesAssociate()
    {
        name = "No record";
        sales = 0;
    }

    public SalesAssociate(String initialName, double initialSales)
    {
        set(initialName, initialSales);
    }

    public void set(String newName, double newSales)
    {
        name = newName;
        sales = newSales;
    }

    public void readInput()
    {
        System.out.print("Enter name of sales associate: ");
        Scanner keyboard = new Scanner(System.in);
        name = keyboard.nextLine();

        System.out.print("Enter associate's sales: $");
        sales = keyboard.nextDouble();
    }

    public void writeOutput()
    {
        System.out.println("Sales associate: " + name);
        System.out.println("Sales:$ " + sales);
    }

    public String getName()
    {
        return name;
    }

    public double getSales()
    {
        return sales;
    }
}
```

图6-4　销售员记录类

程序中将用一个数组来跟踪所有销售员的数据，还需要记录平均销售额与最大销售额。这样，就需要如下的实例变量：

```
private double highest;
private double average;
private SalesAssociate[] record;
```

我们还必须知道销售员的总数，这个值和record.length相同，不过为销售员的总数单独命名一个变量更好些。于是再添加以下实例变量：

```
private int numberOfAssociates; //和record.length一样
```

这样，程序要完成的工作就分解为以下3个子任务：

- 获取数据。
- 计算某些数字（更新实例变量）。
- 显示结果。

那么，程序整体看如下所示：

```
public class SalesReporter
{
    private double highest;
    private double average;
    private SalesAssociate[] record;
    private int numberOfAssociates; //Same as record.length

    public static void main(String[] args)
    {
        SalesReporter clerk = new SalesReporter();
        clerk.getFigures();
        clerk.update();
        clerk.displayResults();
    }
    <还需要更多内容。>
}
```

剩下的工作就是设计getFigures、update及displayResult这3个方法（当然还有测试和调试程序）。下面按顺序逐个展开这3个方法。

因为销售员的类SalesAssociate已有了输入数据的方法，采集输入数据的方法getFigures的实现相对就比较简单。不过，还是有些细微的地方需要考虑。首先设计出一个基本的输入循环：

```
int i;
for (i = 0; i < numberOfAssociates; i++)
{
    System.out.println("Enter data for associate number " + (i + 1));
    record[i].readInput();
}
```

数组索引的编号方式与销售员编号方式不相同，前者从0开始，后者从1开始，上面的代码用record[i]来记录第i + 1号销售员，把这个问题巧妙地解决了。

然而还有一个问题，测试这个循环时，会得到一条错误消息，是关于“空指针”(null pointer)。这个问题之所以存在，是因为record数组的基类型是类。为了看清这个问题，先来考虑其他简单的情形，例如：

```
SalesAssociate a;
a.readInput();
```

这段代码同样会产生“空指针”错误消息。这是因为a仅仅是个名字，它还不是SalesAssociate类的某个对象名。前面的代码忘了使用new来创建对象，相应的代码是这样的：

```
SalesAssociate a;
a = new SalesAssociate();
a.readInput();
```

索引变量record[i]也是类类型变量，因此它也只是个名字，必须用new给record[i]赋值，然后才能用readInput 方法（或任何其他方法）。前面的代码应如下添加：

```
record[i] = new SalesAssociate();
```

添加了这行代码后，getFigures方法的完整定义如图6-5所示。

```
import java.util.*;

/**
 Program to generate sales report.
*/
public class SalesReporter
{
    private double highest;
    private double average;
    private SalesAssociate[] record;//The array object is
                                    //created in getFigures.
    private int numberOfAssociates; //Same as record.length

    public void getFigures()
    {
        Scanner keyboard = new Scanner(System.in);
        System.out.println("Enter number of sales associates:");
        numberOfAssociates = keyboard.nextInt();
        record = new SalesAssociate[numberOfAssociates];
        int i;
        for (i = 0; i < numberOfAssociates; i++)
        {
            record[i] = new SalesAssociate();
            System.out.println("Enter data for associate " + (i+1));
            record[i].readInput();
            System.out.println();
        }
    }

    /**
     Computes the average and highest sales figures.
     Precondition: There is at least one salesAssociate.
    */
    public void update()
    {
        int i;
        double nextSales = record[0].getSales();
        highest = nextSales;
```

main方法在本图中。

在这里创建数组对象。

在这里创建SalesAssociate对象。

图6-5　销售报表程序

```
        double sum = nextSales;
        for (i = 1; i < numberOfAssociates; i++)
        {
            nextSales = record[i].getSales();
            sum = sum + nextSales;
            if (nextSales > highest)
                highest = nextSales; //highest sales figure so far.
        }
        average = sum/numberOfAssociates;
    }
    /**
     Displays sales report on console screen.
    */
    public void displayResults()
    {
        System.out.println("Average sales per associate is $" +
                  average);
        System.out.println("The highest sales figure is $" + highest);
        System.out.println();
        int i;
        System.out.println("The following had the highest sales:");
        for (i = 0; i < numberOfAssociates; i++)
        {
            double nextSales = record[i].getSales();
            if (nextSales == highest)
            {
                record[i].writeOutput();
                System.out.println("$" + (nextSales - average)
                                 + " above the average." );
                System.out.println();
            }
        }

        System.out.println("The rest performed as follows:");
        for (i = 0; i < numberOfAssociates; i++)
        {
            double nextSales = record[i].getSales();
            if (record[i].getSales() != highest)
            {
                record[i].writeOutput();
                if (nextSales >= average)
                    System.out.println("$" + (nextSales - average)
                                     + " above the average.");
                else
                    System.out.println("$" + (average - nextSales)
                                     + " below the average.");
                System.out.println();
            }
        }
    }
    public static void main(String[] args)
```

record[0]已经处理过了，所以循环从record[1]开始。

图6-5 (续)

```
    {
        SalesReporter clerk = new SalesReporter();
        clerk.getFigures();
        clerk.update();
        clerk.displayResults();
    }
}
```

屏幕对话示例

```
Enter number of sales associates:
3
Enter data for associate number 1
Enter name of sales associate: Dusty Rhodes
Enter associate's sales: $36000
Enter data for associate number 2
Enter name of sales associate: Natalie Dressed
Enter associate's sales: $50000
Enter data for associate number 3
Enter name of sales associate: Sandy Hair
Enter associate's sales: $10000
Average sales per associate is $32000
The highest sales figure is $50000
The following had the highest sales:
Name: Natalie Dressed
Sales: $50000
$18000 above the average.
The rest performed as follows:
Name: Dusty Rhodes
Sales: $36000
$4000 above the average.
Name: Sandy Hair
Sales: $10000
$22000 below the average.
```

图6-5 （续）

下一步该实现update方法了。先从下面的代码开始：

```
for (i = 0; i < numberOfAssociates; i++)
{
    sum = sum + record[i].getSales();
    if (record[i].getSales() > highest)
        highest = record[i] .getSales() //highest sales figure
                                        //so far.
}
average = sum/numberOfAssociates;
```

这个循环基本上没问题，不过显然还需要把sum和highest 这两个变量在循环开始之前就初始化。sum可以初始化为0，但是highest要初始化为什么值呢？或许可以用一个负数，因为销售额不应该是负的。可是若顾客退货，就会被当作负的销售额来处理，因此销售额还是可以出现负数的。好在能确信公司至少有一个销售员，这样就可以把sum和highest 都初始化为第一个销售员的业绩。在循环前面增加了相应的处理后，代码如下：

```
highest = record[0].getSales();
double sum = record[0].getSales();
for (i = 1; i < numberOfAssociates; i++)
{
    sum = sum + record[i].getSales();
    if (record[i].getSales() > highest)
        highest = record[i].getSales() ;//highest sales figure
                                         //so far.
}
average = sum/numberOfAssociates;
```

前面的循环能够工作，但是代码中有冗余的计算，就是那3次对record[i].getSales()方法的调用。可以把调用减少为1次，并把结果保存在一个变量里，代码如下：

```
int i;
double nextSales = record[0].getSales();
highest = nextSales;
double sum = nextSales;
for (i = 1; i < numberOfAssociates; i++)
{
    nextSales = record[i].getSales();
    sum = sum + nextSales;
    if (nextSales > highest)
        highest = nextSales;//highest sales figure so far.
}
average = sum/numberOfAssociates;
```

update方法的最终完整版本参见图6-5。

最后一个方法displayResults 的设计只用到了以前见到过的技术，这里就不再详细讲解了。它的最终实现参见图6-5。

自测题

7. 写一段Java代码，声明一个长度为3的数组entry，用图6-4中的SalesAssociate作为基类型。用一个for循环在这个数组中填充3个相同的记录，每条记录的数据都是：名字Jane Doe，销售额$5000。
8. 重写图6-5中的程序SalesReporterRewrite中的displayResults方法，用图5-14中Dollars类中的方法，以正确的美元和美分格式输出总的美元数。

6.2.1 索引变量用作方法的实参

数组a中的索引变量，如a[i]，可以用在和这个数组的基类型相同的普通变量能够使用的任何地方。只要这样的普通变量能用作某个方法的实参，则这个数组的索引变量一样也能用。图6-6中的程序就示范了如何把索引变量作为方法的实参。

average方法有2个实参，类型都是int，而图6-6的程序里的数组nextScore的基类型是int，所以该程序可以用nextScore[i]作为average方法的调用实参，其中的那行代码如下：

```
possibleAverage = average(firstScore, nextScore[i]);
```

变量firstScore 是int类型的普通变量。为了让读者更能体会到索引变量nextScore[i]可以用在任何int类型变量可以使用的地方，注意到如果把average方法的两个实参交换，图6-6

的程序的运行结果不变，我们可以把前面调用average 的代码改为如下所示：

```
possibleAverage = average(nextScore[i], firstScore);
```

```
import java.util.*;

/**
 A program to demonstrate the use of
 indexed variables as arguments.
*/
public class ArgumentDemo
{
    public static void main(String[] args)
    {
        Scanner keyboard = new Scanner(System.in);
        System.out.println("Enter your score on exam 1:");
        int firstScore = keyboard.nextInt();
        int[] nextScore = new int[3];
        int i;
        double possibleAverage;
        for (i = 0; i < nextScore.length; i++)
            nextScore[i] = 80 + 10*i;
        for (i = 0; i < nextScore.length; i++)
        {
            possibleAverage = average(firstScore, nextScore[i]);
            System.out.println("If your score on exam 2 is "
                               + nextScore[i]);
            System.out.println("your average will be "+ possibleAverage);
        }
    }

    public static double average(int n1, int n2)
    {
        return (n1 + n2)/2.0;
    }
}
```

屏幕对话示例

```
Enter your score on exam 1:
80
If your score on exam 2 is 80
your average will be 80.0
If your score on exam 2 is 90
your average will be 85.0
If your score on exam 2 is 100
your average will be 90.0
```

图6-6　索引变量用作实参

同时留意一下average方法的定义，其中没有说明它的实参可以是int数组的索引变量。average 方法只是接受int 类型的实参，它不清楚也不关心这些int的变量是索引变量，还是普通的int变量，或者是int常量值。

索引变量用作方法的实参时，有一处有点儿微妙的地方。例如，下面是一个方法调用：

```
possibleAverage = average(firstScore, nextScore[i]);
```

若i的值是2，实参实际就是nextScore[2]；若i为0，则实参就是nextScore[0]。必须求出

索引表达式的值，以确定究竟是哪一个索引变量被用来当作实参。

必须明确，数组a的索引变量，比如a[i]，是一个类型为该数组的基类型的变量。a[i]用作方法的实参时，与其他任何同样类型的变量一样。尤其是，若数组的基类型是基本类型，如int、double或char这样的类型，被调用的方法就不能修改a[i]的值；反之，若数组基类型是类，则被调用的方法可以修改以a[i]为名称的对象。

只需要记住，一个索引变量，如a[i]，其类型就是该数组的基类型，对它的处理方式与其他任何同样类型的变量一样。

快速参考：数组索引变量用作实参

数组索引变量可以用作实参，使用的方法和场合与任何其他具有该数组基类型的变量一样。例如，有以下数组：

```
double[] a = new double[10];
```

a[3]和a[index]等索引变量可以用在任何接受double 类型变量的地方作为实参。

常见问题：方法何时能够改变索引变量实参？

假设a[i]是数组a的一个索引变量，并且a[i]如下所示被用于方法调用：

```
doStuff(a[i]);
```

doStuff方法是否能修改a[i]取决于数组a的基类型。若数组a的基类型是基本类型，比如int，double或char这样的，那就和任何其他的作为实参的基类型一样，doStuff方法不能改变a[i]的值。但是，如果数组a的基类型是类，则doStuff方法就能修改名为a[i]的对象。注意，索引变量a[i]只是被当作那些其类型是数组的基类型的变量一样看待。

6.2.2 整个数组作为方法的实参

不仅仅索引变量可以用作方法的实参，整个数组也可以作为方法的一个实参。在方法的定义中指定数组形参的方法，与声明数组的方法类似。例如，下面将看到的increment-ArrayBy2 方法就接受任何double 数组作为实参：

```
public class SampleClass
{
    public static void incrementArrayBy2(double[] a)
    {
        int i;
        for (i = 0; i < a.length; i++)
            a[i] = a[i] + 2;
    }
    <这里是剩余的类定义。>
}
```

现在用下面几个数组来演示这种用法：

```
double[] a = new double[10];
double[] b = new double[30];
```

假设数组a和b中的元素都已经正确地赋值了，则下面的代码都是合法的方法调用：

```
SampleClass.incrementArrayBy2(a);
SampleClass.incrementArrayBy2(b);
```

关于整个数组作为实参，有些方面需要强调一下。首先，把整个数组传给方法作为实参时，不需要使用方括号。其次，方法可以改变数组中的值，前面代码中的increment-ArrayBy2方法就已经说明了这一点。第三，可以用不同长度的数组实参来替换同一个数组形参。注意，前面的incrementArrayBy2 可以接受任意长度的数组作为实参。指定一个参数是数组形参时，只能指定数组的基类型，不能指定其长度。

6.2.3 main方法的实参

程序的main方法的头部如下：

```
public static void main(String[] args)
```

String[] args这部分使args看起来是个数组形参，其基类型为String。实际情况也是如此，main 方法接受String 数组值作为实参。但在运行程序时，从来没有给main 传递过数组实参，或其他任何类型的实参，这究竟是怎么回事？

答案是：对main方法的调用是进行特殊处理的。运行程序时，默认的字符串数组被传给main作为其默认的实参。

快速参考：数组形参和数组实参

方法的实参可以是整个数组。数组实参就像类的对象，在该方法中，可以修改数组实参中的值。下面的例子展示了定义和调用具有数组形参的方法。(这里所有的例子代码都假设是在某个类的定义中。)

举例（数组形参）：

```
public static void showArray(char[] a)
{
    int i;
    for (i = 0; 1 < a.length; i++)
        System.out.println(a[i]);
}

public static void reinitialize(int[] anArray)
{
    int i;
    for (i = 0; i < anArray.length; i++)
        anArray[i] = 0;
}
```

举例（数组实参）：

```
char[] symbol = new char[10];
int[] a = new int[10];
int[] b = new int[20];

  <这里是某些填充数组的代码。>

showArray(symbol);
reinitialize(a);
reinitialize(b);
```

注意数组a和b的长度不同。同时也要注意数组实参没有使用方括号。

等价的语法：

尽管我们不推荐使用它，读者还是可能会碰到数组形参的另一种等价语法，例如：

```
public static void showArray(char a[])
```

可以替换下面的代码：

```
public static void showArray(char[] a)
```

运行程序时，还可以提供额外的字符串实参，这些字符串实参都会被自动添加进传递给main方法的数组实参中作为其元素。通常是在操作系统的命令行方式下运行程序时这样做：

```
java TestProgram Sally Smith
```

此时args[0] 被设置为"Sally"，args[1]被设置为"Smith"，并且这两个索引变量可以在main方法中使用，例如：

```
public class TestProgram
{
    public static void main(String[] args)
    {
        System.out.println("Hello " + args[0] + " " + args[1]);
    }
}
```

编译上面的程序，并用一行命令在操作系统中执行：

```
java TestProgram Josephine Student
```

该程序产生的输出如下：

```
Hello Josephine Student
```

因为标识符args是个形参，所以args可以被替换成任何其他非关键字的标识符，其意义不变（当然必须在main方法中把args的每个引用都替换成相应的标识符）。不过，习惯上这个参数使用标识符args。

最后，需要明确的是，传给main的实参是字符串数组。如果需要数字，则必须把数字的字符串表示转换成相应的数字类型的值。

快速参考：main方法有个数组形参

程序的main方法的头部如下：

```
public static void main(String[] args)
```

标识符args是基类型为String的数组形参。详细描述请见正文。

▲ 易犯错误：对数组使用 = 和 = =

所有的数组都是对象，所以赋值操作符 = 和相等操作符 == 在各种各样的对象上的作用（及副作用）也同样适用于数组。为了理解它们对于数组的作用，必须多了解一些数组在计算机主存中的存储方式。讨论的关键点在于整个数组所包含的内容（也就是所有的索引变量）是存储在一整块（可能很大）内存中的，所以整个数组所包含内容的位置可以用一个内存地址来表示。

回想一下，对象的变量实际上保存的是该对象的内存地址。赋值操作符复制的就是这个内存地址。例如：

```
int[] a = new int[3];
int[] b = new int[3];
int i;
for (i = 0; i < a.length; i++)
```

```
        a[i] = i;
    b = a;
    System.out.println("a[2] = " + a[2] + " b[2] = " + b[2]);
    a[2] = 2001;
    System.out.println("a[2] = " + a[2] + " b[2] = " + b[2]);
```

这段代码将产生如下的输出：

```
    a[2] = 2 b[2] = 2
    a[2] = 2001 b[2] = 2001
```

前面这段代码中的赋值语句b=a，使数组变量b中的内存地址和数组变量a里的内存地址相同，因此a和b就是同一个数组的不同名字。这样，改变a[2]的值时，也就是在改变b[2]的值。

上面讲到的状况比较复杂，最简单的办法就是不要对数组使用=（也不要用==）。如果想让前面代码中的数组a和b作为不同的数组而拥有同样的值，下面的赋值语句是不对的：

```
    b = a;
```

而应该用下面这样的代码：

```
    int i;
    for (1 = 0; i < a.length; i++)
        b[i] = a[i];
```

注意，上面的代码假设数组a和b的长度相同。

相等运算符==的作用是测试两个数组是否保存在计算机内存的同一个位置。例如：

```
    int[] a = new int[3];
    int[] b = new int[3];
    int i;
    for (i = 0; i < a.length; i++)
        a[i] = i;
    for (i = 0; i < a.length; i++)
        b[i] = i;
    if (b == a)
        System.out.println("Equal by ==");
    else
        System.out.println("Not equal by ==" );
```

这段代码产生如下输出：

```
    Not equal by ==
```

即使数组a和b在相同索引变量中包含的值相同，这段代码也仍然输出同样的结果，因为数组a和b保存在不同的内存位置，而==只是测试内存地址是否相同。

若需要测试两个数组是否包含同样的元素，则可以为数组定义equals方法，这就像为一个类定义equals方法一样。例如，图6-7为一个小演示类中的数组给出了一个可能的equals方法的定义。△

```
/**
 This is just a demonstration program to see how
 equals and == work.
*/
public class TestEquals
{
    public static void main(String[] args)
    {
        int[] a = new int[3]; int[] b = new int[3]; int i;
        for (i = 0; i < a.length; i++)
            a[i] = i;
        for (i = 0; i < b.length; i++)
```

数组a和b包含同样的一些整数且其顺序也一样。

图6-7　两种相等性

```
            b[i] = i;
        if (b == a)
            System.out.println("Equal by ==.");
        else
            System.out.println("Not equal by ==.");
        if (equals(b,a))
            System.out.println("Equal by the equals method.");
        else
            System.out.println("Not equal by the equals method.");
    }

    public static boolean equals(int[] a, int[] b)
    {
        boolean match;
        if (a.length != b.length)
            match = false;
        else
        {
            match = true; //tentatively
            int i = 0;
            while (match && (i < a.length))
            {
                if (a[i] != b[i])
                    match = false;
                i++;
            }
        }
        return match;
    }
}
```

屏幕输出

```
Not equal by ==.
Equal by the eauqls method.
```

图6-7 （续）

记住：数组类型是引用类型

数组类型的变量只是保存了数组在内存中的地址。这个内存地址经常被称为对内存中的数组对象的**引用**（reference）。因此，数组类型常被称为引用类型。**引用类型**（reference type）是指这种类型的变量中保存的是引用（即内存地址），而不是此命名变量实际对应的值。数组类型和类类型都是引用类型。基本类型不是引用类型。

常见问题：数组到底是不是对象？

数组不属于任何类。还有其他一些特性是类的对象拥有而数组没有的，比如继承（参见第7章）。因此数组是否应该被看作对象不是百分之百明确的。不过，这大体上是个学术论题。在Java中，已经法定数组都是对象。只要Java的文档中说明某些东西是适用于所有对象的，那它就也适用于数组。

6.2.4 返回数组的方法

在Java中，方法可以返回数组。为返回数组的方法指定返回类型，与指定数组形参的做法一样。图6-8所示的程序是由图6-6中的程序略加修改而成的。该程序所作的计算大部分和图6-6所示的程序相同。不过，在这个新版本中，各种可能的平均分由averageArray方法计算并返回分数的数组（类型是double）。

注意，程序中创建了一个新的数组，然后将其返回：

```
double temp = new double[nextScore.length];
<填充数组 temp>
return temp;
```

```
import java.util.*;
/**
 A program to demonstrate a method returning an array.
*/
public class ReturnArrayDemo
{
    public static void main(String[] args)
    {
        System.out.println("Enter your score on exam 1:");
        Scanner keyboard = new Scanner(System.in);
        int firstScore = keyboard.nextInt();
        int[] nextScore = new int[3];
        int i;
        for (i = 0; i < nextScore.length; i++)
            nextScore[i] = 80 + 10*i;
        double[] averageScore;
        averageScore = averageArray(firstScore, nextScore);
        for (i = 0; i < nextScore.iength; i++)
        {
            System.out.println("If your score on exam 2 is " + nextScore[i]);
            System.out.println("your average will be "+ averageScore[i]);
        }
    }

    public static double[] averageArray(int firstScore, int[] nextScore)
    {
        double[] temp = new double[nextScore.length];
        int i;
        for (i = 0; i < temp.length; i++)
            temp[i] = average(firstScore, nextScore[i]);
        return temp;
    }

    public static double average(int n1, int n2)
    {
        return (n1 + n2)/2.0;
    }
}
```

屏幕对话示例和图6-6相同。

图6-8　返回数组的方法

快速参考：返回数组

方法可以返回数组。具体细节基本上与返回其他类型相同。

语法（返回数组的一种常用方式）：

```
public static Base_Type[] Method_Name(Parameter_List)
{
    Base_Type[] temp = new Base_Type[Array_Size]
    Statements_To_Fill_temp
    return temp;
}
```

方法不必是静态的或是公有的，下面这些都是合法的方法头部：

```
public Base_Type[] Method_Name(Parameter_List)
private static Base_Type[] Method_Name(Parameter_List)
private Base_Type[] Method_Name(Parameter_List)
```

举例（假定这段代码位于某个类的定义中）：

```
public static char[] vowels()
{
    char[] newArray = new char[5];
    newArray[0] = 'a';
    newArray[1] = 'e';
    newArray[2] = 'i';
    newArray[3] = 'o';
    newArray[4] = 'u';
    return newArray;
}
```

快速参考：数组类型名

数组的类型名的形式如下：

Base_Type []

不管是声明数组变量、指定数组形参的类型，或者是指定方法的返回类型为数组，都用这种形式。

举例：

```
int[] n = new int[10];
Species [] s = new Species [20];
public static double[] halfAll(int[] arryToBeHalved);
{
        .
        .
        .
```

自测题

9. 下面的代码产生什么样的输出？

```
char[] a = new char[3];
char[] b;
int i;
for (i = 0; 1 < a.length; i++)
```

```
    a[i] = 'a';
b = a;
System.out.println("a[1] = " + a[1] + " b[1] = " + b[1]);
System.out.println("a[2] = " + a[2] + " b[2] = " + b[2]);
b[2] = 'b';
System.out.println("a[1] = " + a[1] + " b[1] = " + b[1]);
System.out.println("a[2] = " + a[2] + " b[2] = " + b[2]);
```

10. 给出一个名为showArray的方法定义，它有一个参数，是一个基类型为char的数组。该方法向屏幕输出一行文本，这行文本由输入参数的数组中的各个字符依序组成。将此方法作为静态方法。为了测试，可以把此方法加入某个类；或者写一个类，在该类的main方法中有一段测试代码，这样更好些。
11. 给出一个名为halfArray的方法定义，它有一个参数，是一个基类型为double的数组。该方法返回一个与输入参数的数组同样长度且基类型也为double的数组，其中每个元素的值都是输入的数组各对应元素的值除以2.0。将此方法作为静态方法。为了测试，可以把此方法加入某个类；或者写一个类，在该类的main方法中有一段测试代码，这样更好些。
12. 下面的方法定义有何错误？

```
public static void doubleSize(int[] a)
{
    a = new int[a.length * 2];
}
```

它可以编译，但是不会像设想的那样工作。

6.3 用数组和类编程

冥冥有手写天书，
彩笔无情挥不已，
流尽人间泪几千，
不能洗去半行字。

——（波斯）奥玛·伽音，《鲁拜集》（菲兹杰拉德（英）译本），中文原创译者为 Kerson Huang（黄克孙）

本节里要介绍和数组相关的进一步的技术，尤其是把数组变量作为类的实例变量。首先用一个编程示例来展示一些基本的技术。

编程示例：专用的列表类

使数组用于某种特定用途的一种方法是把数组用作类的实例变量，然后只允许通过类的方法来访问数组。可以定义类，使其对象像是某种专用的数组，而这种数组只能通过类的方法才能访问，于是就可以添加任何需要的检查和自动处理功能。在这个编程示例中，我们给出这样的一个类作为示范。

定义一个类，其对象可以用于保存一系列条目，比如杂货清单或者待处理事项列表。这个类有个相当长的名字——OneWayNoRepeatsList[①]。

OneWayNoRepeatsList类有个方法可以向列表中添加条目。列表中的条目是字符串，在应用程序

① 在Java中，使用长名字是惯例，不过不会仅为了遵循惯例而去选择长的名字。很多短名字，如List、Table等，在计算机科学中已有了其特定的技术含义，若使用这些短名字却不用其已约定俗成的意义，将会导致混乱。

中可以用来表示任何东西，比如"Buy milk"（买牛奶）。这个类没有提供方法来改变或者删除列表中的单个条目。不过，它提供能把整个列表删除的方法，重新从空列表开始。OneWayNoRepeatsList类的每个对象能够保存的条目数目有上限。任何时候，列表对象中可能含有的条目数在0和这个最大值之间。

OneWayNoRepeatsList类的对象有个字符串数组作为实例变量。该数组就保存了列表中的条目。不过，不允许直接访问该数组，而是要通过专用的访问和设置方法来访问。可以用一些int变量来表示列表中的位置。这种int变量就等同于索引，不过位置是从1开始编号的，而不是从0开始。例如，名为getEntryAt 的方法可以得到指定位置的条目。假设toDoList 是OneWayNoRepeatsList类的一个对象，下面的代码会设置字符串变量next为位置2处的条目：

```
String next = toDoList.getEntryAt(2);
```

没有办法（直接）修改列表中的条目，但有方法可以在列表的末尾添加条目及删除整个列表。不过这些是能改变列表的仅有方式。

在第4章中已经讨论过封装，OneWayNoRepeatsList 类就是一个封装得很好的类。正如第4章中所说的，一个封装良好的类，其定义能让使用这个类的程序员不需要知道该类是如何实现的这样的细节。如果OneWayNoRepeatsList类符合这个要求，就可以在给出这个类的定义之前先来学习如何使用它。下面就从学习OneWayNoRepeatsList类的使用开始。

图6-9中的程序演示了如何使用OneWayNoRepeatsList类中的一些方法。注意，其中有一个构造方法需要一个整数实参，该整数指定了列表中能够容纳的条目数目的最大值。通常，列表中包含的条目数是小于最大值的。

```
import java.util.*;
public class ListDemo
{
    public static void main(String[] args)
    {
        OneWayNoRepeatsList toDoList =
                                new OneWayNoRepeatsList(3);
        System.out.println(
                        "Enter items for the list, when prompted.");
        boolean more = true;
        String next = null, ans = null;
        Scanner keyboard = new Scanner(System.in);

        while ( more && (! toDoList.full()))
        {
            System.out.println("Input an entry:");
            next = keyboard.nextLine();
            toDoList.addItem(next);
            if (toDoList.full())
            {
                System.out.println("List is full.");
            }
            else
            {
                System.out.print("More items for the list?");
                ans = keyboard.nextLine();
                if (ans.trim().equalsIgnoreCase("no"))
                    more = false;
            }
```

图6-9 使用OneWayNoRepeatsList类

```
            }

            System.out.println("The list contains:");
            int position = toDoList.START_POSITION;
            next = toDoList.getEntryAt(position);
            while (next != null)
            {
                System.out.println(next);
                position++;
                next = toDoList.getEntryAt(position);
            }
        }
    }
```

null表明到达了列表的结尾。

屏幕对话示例

```
Enter items for the list, when prompted.
Input an entry:
Buy milk.
More items for the list? yes
Input an entry:
Walk dog.
More items for the list? yes
Input an entry:
Buy milk.
More items for the list? yes
Input an entry:
Write program.
The list is full.
The list contains:
Buy milk.
Walk dog.
Write program.
```

图6-9　（续）

addItem方法向列表中添加一个字符串。例如，下面的代码把next变量命名的字符串添加到toDoList列表中：

```
toDoList.addItem(next);
```

注意，在屏幕对话示例中，可以看到"Buy milk."被添加到列表中两次，但是在列表中只出现了一次。如果被添加的条目已经在列表中，addItem方法就没有什么效果。所以这个列表是没有重复项的。

用一个int变量可以在列表中从头到尾进行遍历。图6-9中展示了这种技术。下面的代码把一个int变量初始化为列表第一个项的位置：

```
int position = toDoList.START_POSITION;
```

已定义常量toDoList.START_POSITION的值为1，用这个名字，是因为认为它是列表的起始位置，而不是数字1。可以用getEntryAt方法得到列表中某个指定位置的项。例如，下面的代码把字符串变量next设置为等于变量position所给定的位置（也就是索引）处的字符串：

```
next = toDoList.getEntryAt(position);
```

为了得到列表中的下一个项，程序只是简单地把position的值增1。下面的代码来自图6-9，演示

如何遍历列表：

```
int position = toDoList.START_POSITION;
next = toDoList.getEntryAt(position);
while (next != null)
{
    System.out.println(next);
    position++;
    next = toDoList.getEntryAt(position);
}
```

当position的值增加到超过列表中最后那项的位置时，position相应的索引在数组中就没有对应的项了。因此，我们需要方便地指示出已经到达列表结尾，否则就会访问数组中没有使用过的部分——"垃圾值"。为了仔细地处理这个问题，我们定义toDoList.getEntryAt(position)，使给定的位置没有项时，返回值为null。注意，null与任何实际的字符串都不相同，因此null不会作为项出现在任何列表中。于是，程序可以通过检查值是否是null来判断是否到达列表的结尾。前面介绍过，为了测试与null的相等性或者不等性，只需要用==或者!=，不需要使用equals方法。

OneWayNoRepeatsList类的完整定义在图6-10中给出。列表中的项保存在实例变量entry中，它是一个基类型为String的数组。于是，列表中能够容纳的项最多是entry.length。不过，列表通常并没有充满，所以所含的项数就小于entry.length。为了记录当前有多少项被使用，该类就用了一个实例变量countOfEntries，而项本身保存在索引变量entry[0]、entry[1]、entry[2]等，直至entry[countOfEntries-1]中。数组中索引等于或大于countOfEntries的元素是垃圾值，不代表列表中的任何项。因此，要遍历列表时，到entry[countOfEntries-1]之后就要结束了。

```
/**
 An object of this class is a special kind of list. The list can be written only from
 beginning to end. You can add to the end of the list, but you cannot change
 individual entries. You can erase the entire list and start over. No entry may
 appear more than once on the list. You can use int variables as position markers
 into the list. Position markers are similar to array indices, but are numbered
 starting with 1.
*/
public class OneWayNoRepeatsList
{
    public static int START_POSITION = 1;
    public static int DEFAULT_SIZE = 50;

    //entry.length is the total number of items you have room
    //for on the list. countOfEntries is the number of items
    //currently on the list.
    private int countOfEntries; //can be less than entry.length.
    private String[] entry;

    public OneWayNoRepeatsList(int maximumNumberOfEntries)
    {
        entry = new String[maximumNumberOfEntries];
        countOfEntries = 0;
    }

    /**
     Creates an empty list with a capacity of DEFAULT_SIZE.
```

图6-10 包装在类中的数组

```
    */
    public OneWayNoRepeatsList()
    {
        entry = new String[DEFAULT_SIZE];
        countOfEntries = 0;
    }

    public boolean full()
    {
        return (countOfEntries == entry.length);
    }

    public boolean empty()
    {
        return (countOfEntries == 0);
    }

    /**
     Precondition: List is not full.
     Postcondition: If item was not on the list,
     it has been added to the list.
    */
    public void addItem(String item)
    {
        if (! onList(item))
        {
            if (countOfEntries == entry.length)
            {
                System.out.println("Adding to a full list!");
                System.exit(0);
            }
            else
            {
                entry[countOfEntries] = item;
                countOfEntries++;
            }
        }//else do nothing. Item is already on the list.
    }

    /**
     If the argument indicates a position on the list, then the entry
     at that specified position is returned; otherwise, null is returned.
    */
    public String getEntryAt(int position)
    {
        if ((1 <= position) && (position <= countOfEntries))
            return entry[position - 1];
        else
            return null;
    }

    /**
```

图6-10 （续）

```
        Returns true if position is the index of the
        last item on the list; otherwise, returns false.
       */
       public boolean atLastEntry(int position)
       {
           return (position == countOfEntries);
       }


       /**
        Returns true if item is on the list;
        otherwise, returns false. Does not differentiate
        between upper- and lowercase letters.
       */
       public boolean onList(String item)
       {
           boolean found = false;
           int i = 0;
           while ((! found) && (i < countOfEntries))
           {
               if (item.equalsIgnoreCase(entry[i]))
                   found = true;
               else
                   i++;
           }

           return found;
       }

       public int maximumNumberOfEntries()
       {
           return entry.length;
       }

       public int getNumberOfEntries()
       {
           return countOfEntries;
       }

       public void eraseList()
       {
           countOfEntries = 0;
       }
   }
```

图6-10 （续）

例如，在onList方法的实现中，有一个while循环遍历数组，检查实参是否和列表中的某项相同。代码只检查那些索引小于countOfEntries的元素，并没有检查整个数组，因为数组中索引大于或等于countOfEntries的项不会“在列表中”。因此检查item是否在列表中的while循环代码如下：

```
while ((! found) && (i < countOfEntries))
{
    if (item.equalsIgnoreCase(entry[i]))
        found = true;
```

```
    else
        i++;
}
```

除了在图6-9中的演示程序所用到的方法之外，OneWayNoRepeatsList 类还有其他一些方法，使该类能够在很广泛的应用程序中发挥更大的作用。

注意，尽管数组的索引从0开始，如果要用一个int类型的变量作为位置标志，比如图6-1中的变量position，它的计数是从1开始的，不是0。类的方法会自动调整索引，因此当需要位于position处的项时，就会给出entry[position - 1]。

OneWayNoRepeatsList 类提供了3种方法检测列表的结尾。首先，如果在列表中用一个int类型的变量position遍历列表，那么当position和getNumberOfEntries()相等时，就已经到了最后一项了。其次，若atLastEntry(position)返回true，说明position 处于列表的最后项。最后，若position不断增加以致超出了最后一项，则getEntryAt(position)返回null。

6.3.1　部分填充的数组

图6-10中的OneWayNoRepeatsList类里的entry数组，是作为部分填充的数组而使用的。在某些情况下，只需要数组中的某些但不是全部的索引变量，例如数组entry包含了一些列表项，但列表并没有充满。在这种情况下，需要记录有多少数组已经被使用了，以及还有多少未被使用。通常用一个int类型变量就可以做到这一点，例如在图6-10中OneWayNoRepeatsList类的实例变量countOfEntries。该变量表明，列表中包含的项在数组的索引0到countOfEntries－1之间，图6-11中对此给出了图示。记录数组中当前有多少项已被使用非常重要，因为其他的数组项中包含的都是一无是处的垃圾值。当访问部分填充的数组时，只需要访问数组的前面包含了有效数据的元素即可，并且要忽略数组的剩余部分。当然，随着对部分填充的数组增加或删除项，有效的值和垃圾值之间的分界线会移动。通过改变某个合适的int类型变量的值，比如countOfEntries，这种移动就能被记录下来。

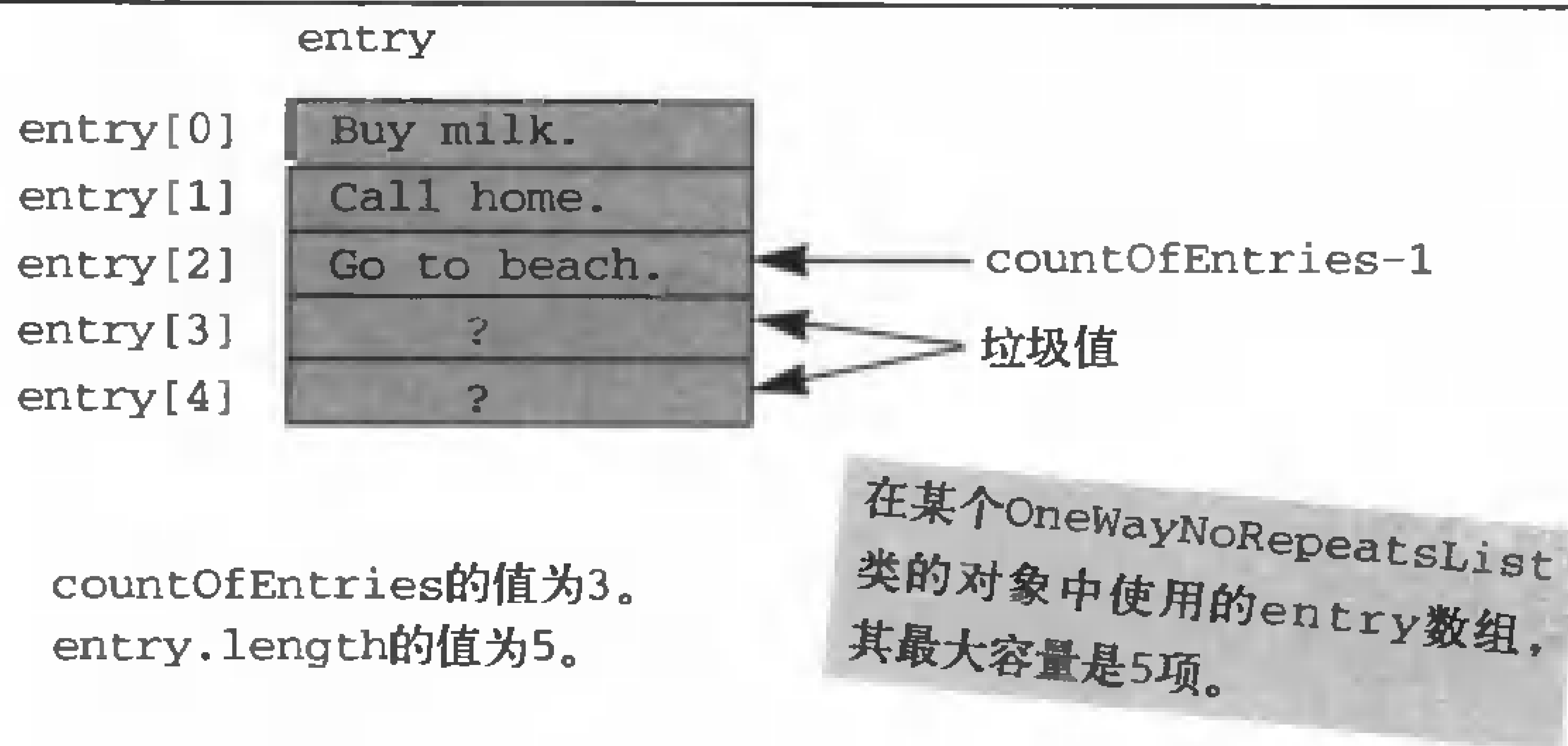


图6-11　部分填充的数组

6.3.2　在数组中查找

图6-10中的类OneWayNoRepeatsList有个方法onList，它在数组entry中查找是否其中有元素和参数item相等。这是对数组进行**顺序搜索**（sequential search）的一个例子。顺序搜索算法很简单也很直截了当：代码从头到尾依顺序逐个查看数组元素是否和要找的数组元素

项相等。（如果数组是部分填充的，查找进行到有效值的最末一个时停止。）

▲ 易犯错误：返回数组实例变量

有人想把下列访问方法加入图6-10所示的OneWayNoRepeatsList类中：

```
public String[] getEntryArray()
{
    return entry;
}
```

阅读正文了解这个定义错在哪里。

这个定义看起来没有什么问题。它与其他为了读取私有的实例变量而提供的访问方法很相像。然而，实际上由于返回的是一个数组，而不是基本类型如int或double，这使得情况很不相同。该方法返回了数组，数组变量实际所含的是引用（即内存地址），这就意味着该方法提供了一种途径能绕过private私有指示。

假设给OneWayNoRepeatsList类添加了方法getEntryArray，并且该方法的定义如本节开头所示。进一步假设程序员如下所示创建了OneWayNoRepeatsList类型的对象：

```
OneWayNoRepeatsList myList = new OneWayNoRepeatsList();
```

现在若该程序员想直接访问myList对象的私有数组实例变量entry，应当无法获得其访问权，因为在类定义中，entry前面的修饰符private就是被明确地设计为禁止这类访问的。然而，为了获得对私有的实例变量myList.entry的访问权，这个程序员（也可能只是太粗心大意）只需要使用如下代码：

```
String[] a = myList.getEntryArray();
```

执行上面代码之后，数组变量a含有数组myList.entry的地址。这样a成了私有数组myList.entry的别名！该程序员只要用名字a，就可以对私有的数组myList.entry进行任何操作。比如：

```
myList.entry[2] = "Party tonight!";
```

因为实例变量myList.entry是私有的，程序员不能用entry这个名字，上述代码会产生编译错误消息。但是，如果能像下面这样写，就意味着是完全相同的东西，因为a是myList.entry的别名：

```
a[2] = "Party tonight!";
```

数组myList.entry和a是同一个数组！

显然，上述的访问方法getEntryArray的定义不能算是个合适的访问方法。那么如何才能为数组entry提供访问方法呢？第一个答案是除了那些已经有的方法，根本不需要有别的访问方法（或设置方法）。getEntryAt、full和empty等这些方法已经允许程序员对OneWayNoRepeatsList对象进行任何合法的操作了。实际上，极少需要一个访问方法来返回整个数组实例变量。然而，不需要并不意味着办不到。

假设因为某种原因，需要给OneWayNoRepeatsList类添加访问方法getEntryArray，其正确的实现如下：

```
public String[] getEntryArray()
{
    String[] temp = new String[entry.length];
    int i;
    for (i = 0; i < countOfEntries; i++)
        temp[i] = entry[i];
    return temp;
}
```

该方法定义中构造的temp数组，含有和数组entry一样的一些项，但是它是个不同的数组。它是数组entry的一个副本，不是数组entry本身。使用getEntryArray的这个新定义，下面的代码就彻底

安全了：

```
String[] a = myList.getEntryArray();
```

程序员可以修改数组a，但是任何修改都不会影响数组myList.entry。数组a是myList.entry的一个等价副本，但不是同一个数组①。

若所考虑的数组的基类型是基本类型或String类型，事情基本如此。然而，如果数组的基类型是类，问题还没这么容易就解决。如果私有实例数组变量的基类型是类，而且需要访问方法返回该数组的一个安全副本，那么不仅必须复制数组，而且要复制数组中的每个项。不过，未必需要对基类型是类的数组提供访问方法返回整个数组的副本（除非是String类，在此处的讨论中，它可以被当作基本类型）。因此，不再继续纠缠这一点了。这个问题实质上就是第5章的易犯错误“隐私泄露”部分中讨论的问题。 △

自测题

13. 假设a是类型为double的数组。编写代码把a中的所有元素显示在屏幕上，每行一个元素。
14. 假设a是类型为double的数组。它是个部分填充的数组，仅在第一个numberUsed个元素中含有有意义的值。（numberUsed是个int类型的变量，其值表示数组中有意义的元素的个数。）编写代码显示数组a中所有有意义的元素。
15. 假设a是部分填充的数组，容量为10，不过只含有3个元素。哪些索引变量含有这3个元素？
16. 对于自测题14中的那个数组a。编写代码将数字42添加到部分填充的数组a中。（提示：必须更新numberUsed。可以假设数组a并没有充满。）
17. 重做自测题16，不过假设不知道数组a是否充满。如果a是满的，所写的代码应当向屏幕上输出适当的消息。

6.4 对数组进行排序

这是个包容万物的地方，每样东西都各得其所。

——伊莎贝拉·玛莉·彼顿，美国女作家，《居家管理》

假设有一个数组，需要对其中的值按照某种方式排序。比如，需要对一个数字数组按照从小到大或者从大到小排序，或者需要对一个字符串的数组按照字母表顺序排序。本节将介绍一种简单的排序算法并给出该算法的Java实现。我们将使该算法适用于对基类型为int的数组进行排序。不过，只需要很少的、显而易见的修改，它就可以适用于对具有任何可排序的基类型的数组进行排序，例如含有表示雇员记录的对象的数组需要按社会保险号码排序。

6.4.1 选择排序

本小节介绍**选择排序**（selection sort），它是最容易理解的排序算法之一。

该算法将被实现为一个方法，有一个基类型为int的数组形参a，就是要被排序的数组。该方法会重新安排索引变量中的值：

```
a[0] <= a[1] <= a[2] <= ... <= a[a.length - 1]
```

① 可能会有疑义说这两个数组并不等价，因为没有对部分填充的数组后面的垃圾值进行复制。不过，这些只是无用的值而已，况且，若确实想这么做，也可以复制垃圾值，但是实在是没有理由这样做。

选择排序算法几乎能从我们要求该算法做什么的指示中自动得出。我们需要该算法重新排列数组中的值，使a[0]是最小的，a[1]是次小的，依次类推。这种指示就得到了以下算法概要：

```
for (index = 0; index < a.length; index++)
    将第index个最小的元素放在a[index]中
```

(在这里，从0开始计数，所以最小的元素称为第0个最小的元素，下一个元素称为第1个最小的元素，依次类推。)

下面只有一个数组a来实现这个算法的细节。这意味着没有额外的空间供用户存储正在移动的元素。移动数组中的元素而不丢失任何元素的唯一办法就是让元素交换位置。任何采用了交换值的方式进行排序的算法都称为**交换排序算法**（interchange sorting algorithm）。按照这样的定义，选择排序算法属于交换排序算法。下面从一个例子开始，看看数组的元素是如何交换的。

图6-12显示了如何通过交换值来对数组排序。第一个数组的图表示开始的值。数组中最

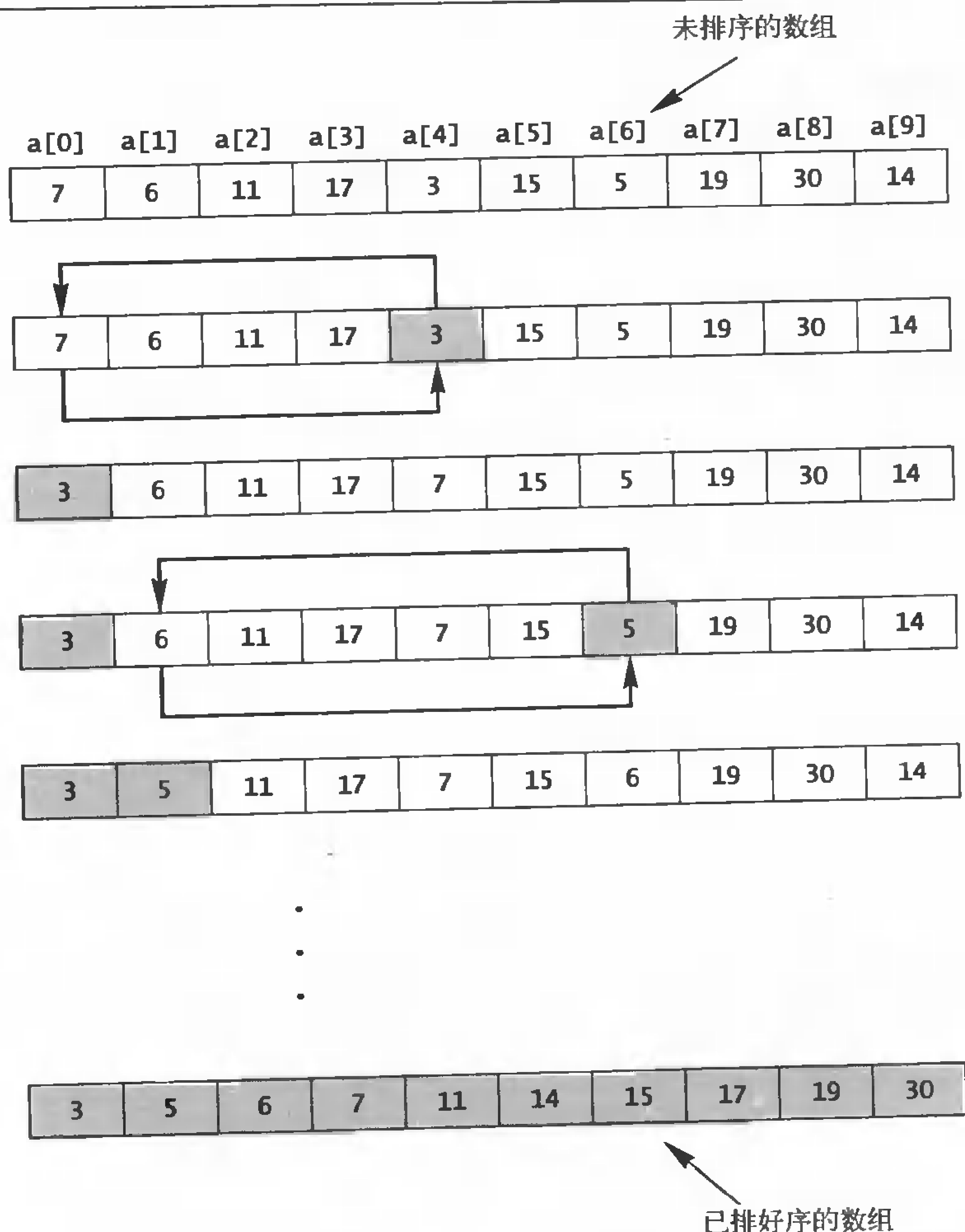


图6-12 通过交换值进行排序

小的值是a[4]中的3。因此a[4]的值需要和a[0]的值交换。交换之后，最小的值在a[0]中。第二个最小值是a[6]中的5，所以a[6]中的值需要和a[1]中的值交换。此后，a[0]和a[1]中的值就是最小的和第二小了，这就是它们在最终排序后的数组中应当在的位置。该算法继续把下面一个最小值和a[2]中的值交换，依次类推，直到整个数组排好序。

这个分析过程就得出了下面的选择排序算法的伪代码。

选择排序算法对数组a排序

```
for (index = 0; index < a.length - 1; index++)
{//在a[index]中放入正确的值：
    indexOfNextSmallest = 下列元素中最小值所在的索引
                a[index], a[index+1], ..., a[a.length-1];
    将a[index]中的值和a[indexOfNextSmallest]中的值交换。
    //a[0] <= a[1] <= ... <= a[index]并且它们是原始数组中最小的元素。
    //余下的位置中含有原始数组中除了这些值之外的其他元素。
}
```

注意，for循环在正确填充a[a.length-2]之后就结束了，尽管最后的索引是a.length-1。这样做没有问题，因为此时只剩下一个元素需要被交换到位，也就是a[a.length-1]，它一定已经正确就位了。为了看出这一点，注意当算法把除a[a.length-1]之外的部分都正确地排序之后，a[a.length-1]应当放置的值就是剩下需要移动的最小的值，而仅剩的需要移动的值已经位于a[a.length-1]中了。

图6-13中包含了一个带有静态方法名为sort的类，它实现了这个选择排序算法。sort方法使用了两个私有的辅助方法，名为indexOfSmallest和interchange。一旦理解了indexOfSmallest和interchange方法，就很容易看出sort方法的定义是直接把前面的伪代码翻译为了Java代码。因此先来研究这两个方法。

```
/**
 Class for sorting an array of base type int from smallest to largest.
*/
public class SelectionSort
{
    /**
     Precondition: Every indexed variable of a has a value.
     Action: Sorts a so that a[0] <= a[1] <= ... <= a[a.length - 1].
    */
    public static void sort(int[] a)
    {
        int index, indexOfNextSmallest;
        for (index = 0; index < a.length - 1; index++)
        {//Place the correct value in a[index]:
            indexOfNextSmallest = indexOfSmallest(index, a);
            interchange(index,indexOfNextSmallest, a);
             //a[0] <= a[1] <=...<= a[index] and these are the smallest
            //of the original array elements. The remaining positions
            //contain the rest of the original array elements.
        }
    }

    /**
```

图6-13　选择排序类

```
     Returns the index of the smallest value among
     a[startIndex], a[startIndex+1], ... a[a.length - 1]
    */
    private static int indexOfSmallest(int startIndex, int[] a)
    {
        int min = a[startIndex];
        int indexOfMin = startIndex;
        int index;
        for (index = startIndex + 1; index < a.length; index++)
            if (a[index] < min)
            {
                min = a[index];
                indexOfMin = index;
                //min is smallest of a[startIndex] through a[index]
            }
        return indexofMin;
    }
    /**
     Precondition: i and j are valid indices for the array a.
     Postcondition: Values of a[i] and a[j] have been interchanged.
    */
    private static void interchange(int i, int j, int[] a)
    {
        int temp;
        temp = a[i];
        a[i] = a[j];
        a[j] = temp; //original value of a[i]
    }
}
```

图6-13 （续）

indexOfSmallest方法在下列数组元素中查找，并返回其中最小元素的索引：

```
a[startIndex], a[startIndex+1] , ... , a[a.length-1]
```

该方法的实现使用了两个局部变量min和indexOfMin。在查找过程中的任何时刻，min等于迄今为止已发现的数组中的最小值，而indexOfMin就是那个值所在的索引。于是，不管怎样，a[indexOfMin]的值就是min。刚一开始的时候，min设置为a[startIndex]，那是所考察的第一个值，并且将indexOfMin设置为startIndex。接下来，逐个检查每个数组元素，看看是不是更小。当所有候选的数组元素都被检查过之后，该方法返回indexOfMin的值。

名为interchange的方法交换a[i]同a[j]的值。其定义中有个微妙之处，若执行下面的代码：

```
a[i] = a[j];
```

a[i]中原先保存的值就会丢失。因此在此之前，a[i]中的值被保存在了临时变量temp中。

图6-14给出了一个演示程序，说明实际如何使用这个选择排序类。

有许多著名的排序算法，其中有不少比选择排序更高效（也更加复杂）。不过，作为对一般的排序的介绍，选择排序足以胜任了。

```
public class SelectionSortDemo
{
    public static void main(String[] args)
    {
        int[] b = {7, 5, 11, 2, 16, 4, 18, 14, 12, 30};

        System.out.println("Array values before sorting:");
        int i;
        for (i = 0; i < b.length; i++)
            System.out.print(b[i] + " ");
        System.out.println();
        SelectionSort.sort(b);
        System.out.println("Array values after sorting:");
        for (i = 0; i < b.length; i++)
            System.out.print(b[i] + " ");
        System.out.println();
    }
}
```

屏幕输出

```
Array values before sorting:
7 5 11 2 16 4 18 14 12 30
Array values after sorting:
2 4 5 7 11 12 14 16 18 30
```

图6-14　SelectionSort类的演示

6.4.2　其他排序算法

选择排序算法不是最有效的排序算法。实际上，它比许多著名的排序算法要低效得多。然而，选择排序比其他排序算法要简单得多。在编码实现的时候，越简单的算法越不容易出错。因此，如果急需用编码实现一个排序算法，用选择排序（或者其他的简单算法）更加稳妥。

另一方面，如果效率是主要问题，可能就需要用一个更高效也更复杂的算法。但是要知道更复杂的算法需要更长时间来编码、测试和纠错。效率是个精深的话题。要牢记一点，得到错误的结果总是毫无效率的，不管程序能多快得到这个结果。

本章结尾的编程项目4和5描述了另外两种排序算法。本书姊妹篇《Java程序设计与问题解决：高级篇（第4版）》讨论递归的时候，将介绍更高效的排序算法。

自测题

18. 怎样使用SelectionSort类对下面的数组排序？

```
int[] myArray = {9, 22, 3, 2, 87, -17, 12, 14, 33, -2};
```

19. 如何修改SelectionSort类，使其能对double（而不是int）类型值的数组排序？
20. 如何修改SelectionSort类使其能对int类型值的数组进行降序排序，而不是升序？
21. 若一个int类型值的数组中有某个值出现了两次（如b[0]==7且b[5]==7），用SelectionSort.sort方法对这个数组排序。当排序结束之后，这个重复的值会有一份还是有两份副本呢？

6.5 多维数组

不要相信一般印象，我的孩子，
但要关注细节。

——亚瑟·柯南道尔爵士，《身份案件》（夏洛克·福尔摩斯系列）

数组具有多个索引有时候挺有用。例如，假设要把图6-15中的表格保存在某种数组中。表格中未加灰底的部分只是标签，加灰底的部分是实际的项，总共有60个项。如果使用只有一个索引的数组，该数组的长度将是60，每个项的索引数字是多少将很难跟踪。另一方面，若允许有两个索引，就可以把一个索引用于行，另一个用于列，如图6-16所示。

各种不同的复合年利率下的账户余额
（四舍五入到整美元数）

年	5.00%	5.50%	6.00%	6.50%	7.00%	7.50%
1	$1050	$1055	$1060	$1065	$1070	$1075
2	$1103	$1113	$1124	$1134	$1145	$1156
3	$1158	$1174	$1191	$1208	$1225	$1242
4	$1216	$1239	$1262	$1286	$1311	$1335
5	$1276	$1307	$1338	$1370	$1403	$1436
6	$1340	$1379	$1419	$1459	$1501	$1543
7	$1407	$1455	$1504	$1554	$1606	$1659
8	$1477	$1535	$1594	$1655	$1718	$1783
9	$1551	$1619	$1689	$1763	$1838	$1917
10	$1629	$1708	$1791	$1877	$1967	$2061

图6-15 一个数值表格

注意，和简单数组一样，这些索引也是从0开始编号的，而不是从1开始编号。图6-16中也展示了有多个索引的数组元素的Java书写方法。若数组名为`table`且有两个索引，则`table[3][2]`就是那个行号是3而列号是2的元素。恰好有两个索引的数组可以用纸上的二维表格来表示，称为**二维数组**（two-dimensional array）。通常把第一个索引当作行而第二个索引当作列。更一般地说，若数组有n个索引，就称为**n维数组**（n-dimensional array）。这样，我们目前为止用的普通索引数组就是一维数组了。

6.5.1 多维数组基础

有多个索引的数组，其用法和只有一个索引的数组在很大程度上是相同的。为了展示细节，我们将详细分析一个Java例子程序，它用来显示一个如图6-16所示的数组。该程序在图6-17中列出。数组名为`table`，下面的代码声明了这个名字`table`，并创建了该数组：

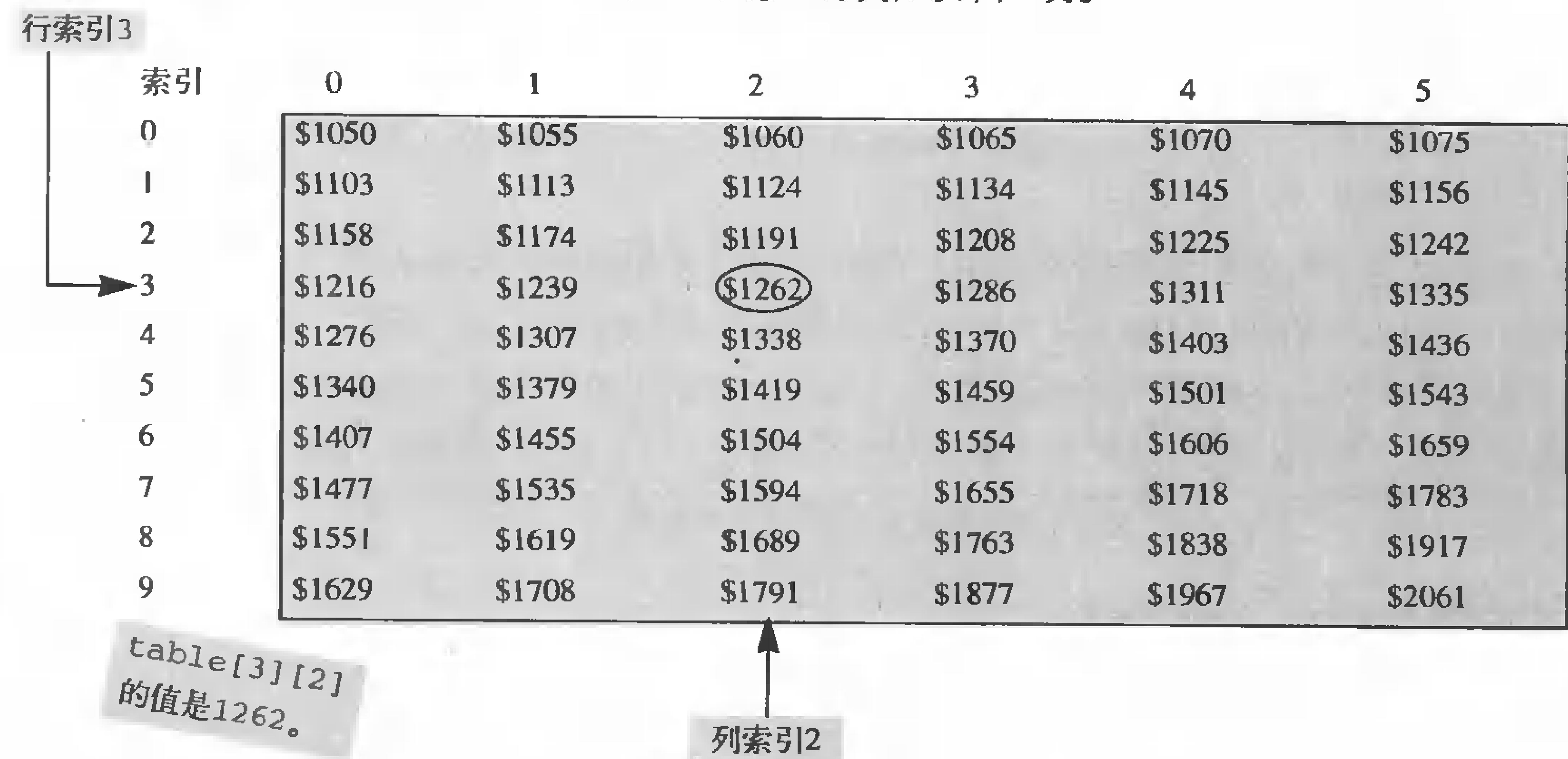
```
int[][] table = new int[10][6];
```

这和下列分两步的代码是等价的：

```
int[][] table;
table = new int[10][6];
```

注意，这在语法上和一维数组的情形几乎相同，唯一的差别是两处都有第二对方括号，并且

对第二维用一个数字规定了其大小（也就是第二个维度中索引的数目）。可以让数组有任意个数的索引，为了有更多的索引，只需要用更多的方括号来声明。



索引	0	1	2	3	4	5
0	$1050	$1055	$1060	$1065	$1070	$1075
1	$1103	$1113	$1124	$1134	$1145	$1156
2	$1158	$1174	$1191	$1208	$1225	$1242
3	$1216	$1239	$1262	$1286	$1311	$1335
4	$1276	$1307	$1338	$1370	$1403	$1436
5	$1340	$1379	$1419	$1459	$1501	$1543
6	$1407	$1455	$1504	$1554	$1606	$1659
7	$1477	$1535	$1594	$1655	$1718	$1783
8	$1551	$1619	$1689	$1763	$1838	$1917
9	$1629	$1708	$1791	$1877	$1967	$2061

图6-16 一个名为table的数组的行和列索引

```
/**
 Displays a two-dimensional table showing how interest
 rates affect bank balances.
*/
public class InterestTable
{
    public static void main(String[] args)
    {
        int[][] table = new int [10][6];
        int row, column;
        for (row = 0; row < 10; row++)
            for (column = 0; column < 6; column++)
                table[row][column] =
                    balance(1000.00, row + 1, (5 + 0.5*column));
        System.out.println("Balances for Various Interest Rates");
        System.out.println("Compounded Annually");
        System.out.println("(Rounded to Whole Dollar Amounts)");
        System.out.println("Years 5.00% 5.50% 6.00% 6.50% 7.00% 7.50%");
        System.out.println();
        for (row = 0; row < 10; row++)
        {
            System.out.print((row + 1) + "       ");
            for (column = 0; column < 6; column++)
                System.out.print("$" + table[row][column] + " ");
            System.out.println();
        }
    }

    /**
     Returns the balance in an account that starts with startBalance
     and is left for the indicated number of years with rate as the
```

真正的应用程序应当对table数组进行更多的处理，这里只是一个演示程序。

图6-17 使用二维数组

```
        interest rate. Interest is compounded annually. The balance is
        rounded to a whole number.
       */
       public static int balance(double startBalance, int years, double rate)
       {
           double runningBalance = startBalance;
           int count;
           for (count = 1; count <= years; count++)
               runningBalance = runningBalance*(1 + rate/100);
           return (int) (Math.round(runningBalance));
       }
   }
```

屏幕对话示例

```
Balances for Various Interest Rates
Compounded Annually
(Rounded to Whole Dollar Amounts)
Years    5.00%   5.50%   6.00%   6.50%   7.00%   7.50%

1        $1050   $1055   $1060   $1065   $1070   $1075
2        $1103   $1113   $1124   $1134   $1145   $1156
3        $1158   $1174   $1191   $1208   $1225   $1242
4        $1216   $1239   $1262   $1286   $1311   $1335
5        $1276   $1307   $1338   $1370   $1403   $1436
6        $1340   $1379   $1419   $1459   $1501   $1543
7        $1407   $1455   $1504   $1554   $1606   $1659
8        $1477   $1535   $1594   $1655   $1718   $1783
9        $1551   $1619   $1689   $1763   $1838   $1917
10       $1629   $1708   $1791   $1877   $1967   $2061
```

最后一行没有对齐，因为10有两位数字。这很容易改正，不过这些和数组本身不相关的事情会给这里的讨论添乱。

图6-17 （续）

多维数组的索引变量与一维数组的索引变量类似，只是它们有多个索引，每个都括在一对方括号中。下面的for循环摘自图6-17：

```
for (row = 0; row < 10; row++)
    for (column = 0; column < 6; column++)
        table[row][column] =
            balance(1000.00, row + 1, (5 + 0.5*column));
```

上述语句用了两个for循环，其中一个嵌套在另一个里面。这是逐个遍历二维数组的所有索引变量的常用方法。若有3个索引，则需要用3个嵌套的for循环，对更多的索引就依次类推。图6-16中的图或许有助于理解table[row][column]中索引的意义，以及嵌套的for循环的意义。

和一维数组一样，多维数组的索引变量是基本类型的变量，可以用在相应的基本类型的变量允许使用的任何地方。例如，对于图6-17中的二维数组table，其中的某个索引变量，如table[3][2]，是一个int类型的变量，可以被用在任何普通int类型的变量可以使用的地方。

快速参考：声明和创建多维数组

声明和创建多维数组，与创建和命名一维数组基本相同。只是要用和索引数同样多的方括号而已。

语法：

Base_Type []...[]*Array_Name* = new *Base_Type*[*Length_1*]...[*Length_n*];

举例：

```
char[][] page = new char[100][80];
int[][] table = new int[10][6];
double[][][] threeDPicture = new double[10][20][30];
SomeClass[][] entry = new SomeClass[100][80];
```

这里SomeClass是一个类。

6.5.2 多维数组形参和返回值

方法可以有多维数组形参，也可以返回多维数组作为返回值。使用方法和一维数组的情形类似，只是要用更多的方括号。图6-18中示范了以二维数组作为形参，该程序是在图6-17中程序的基础上进行少量改写得到的。注意那个数组形参的类型是int[][]。

```
/**
 Displays a two-dimensional table showing how interest
 rates affect bank balances.
*/
public class InterestTable2
{
    public static void main(String[] args)
    {
        int[][] table = new int[10][6];
        int row, column;
        for (row = 0; row < 10; row++)
            for (column = 0; column < 6; column++)
                table[row][column] =
                    balance(1000.00, row+1, (5 + 0.5*column));
        System.out.println("Balances for Various Interest Rates");
        System.out.println("Compounded Annually");
        System.out.println("(Rounded to Whole Dollar Amounts)");
        System.out.println("Years 5.00% 5.50% 6.00% 6.50% 7.00% 7.50%");
        System.out.println();
        showTable(table);
    }

    /**
     Precondition: The array displayArray has 10 rows and 6 columns.
     Postcondition: The array contents are displayed with dollar signs.
    */
    public static void showTable(int[][] displayArray)
    {
        int row, column;
        for (row = 0; row < 10; row++)
        {
            System.out.print((row + 1) + "     ");
            for (column = 0; column < 6; column++)
                System.out.print("$" + displayArray[row][column] + " ");
            System.out.println();
        }
    }

    public static int balance(double startBalance, int years, double rate)
     <balance的定义的其他部分和图6-17相同。>
}
```

在本章后面将给出一个更好的showTable的定义。

程序的输出和图6-17相同。

图6-18　多维数组形参

若需要返回多维数组，类型描述方法和用多维数组作形参是一样的。例如，下面的方法返回一个基类型是double的二维数组：

```
/**
 Precondition: Each dimension of startArray is at least
 the value of size.
 The array returned is the same as the size-by-size
 upper left corner of the array startArray.
*/
public static double[][] corner(double[][] startArray, int size)
{
    double[][] temp = new double[size][size];
    int row, column;
    for (row = 0; row < size; row++)
        for (column = 0; column < size; column++)
            temp[row][column] = startArray[row][column];
    return temp;
}
```

6.5.3 多维数组的实现

在Java中，多维数组是用一维数组实现的。以下面的数组为例：

```
int[][] table = new int[10][6];
```

数组table实际上是一个长度为10的一维数组，其基类型是int[]，换句话说，多维数组是数组的数组。

通常情况下，不需要考虑多维数组是数组的数组这一事实。这种细节是由编译器自动处理的。然而，有些情况下知道这种细节是有益的。例如，假设需要写for循环来给二维数组填充值，在图6-18所示的程序中，用常量6和10来控制for循环。若使用实例变量length来控制for循环，代码的风格会更好。然而使用length实例变量时，就是把多维数组当作数组的数组来思考了。比如，下面的代码是对图6-18中main方法里面嵌套的for循环改写而得到的：

```
for (row = 0; row < table.length; row++)
    for (column = 0; column < table[row].length; column++)
        table[row][column] =
                     balance(1000.00, row + 1, (5 + 0.5*column));
```

这里的嵌套for循环需要进一步详细分析。数组table是用如下方法创建的：

```
int[][] table = new int[10][6];
```

这就意味着table实际上是一个长度为10的一维数组，table[0]到table[9]这些索引变量，每个都是基类型为int且长度为6的一维数组。因此第一个for循环到table.length结束。对table这样的二维数组来说，length是第一维的索引的数目，等价于行的数目，在这个例子中就是10。现在继续考察第二个for循环。

二维数组table的第0行是一维数组table[0]，它有table[0].length项。更一般地说，table[row]是个基类型为int的一维数组，有table[row].length项。因此第二个for循环到table[row].length结束。当然，在这个例子中，table[0].length、table[1].length、...、table[9].length都恰好等于6。

可以用多维数组是数组的数组这一事实来改写图6-18中的showTable方法。注意，在图6-18中，showTable方法假设它的数组形参有10行6列，对特定的程序是可以的，但是

showTable的更好定义应该可以用于任何二维数组。在图6-19里，对showTable重新进行了定义，使它的参数可以是基类型为int的任何二维数组，不管其行列数是多少都可以。

```
/**
 The array displayArray can have any dimensions.
 Postcondition: The array contents are displayed with dollar signs.
*/
public static void showTable(int[][] displayArray)
{
     int row, column;
     for (row = 0; row < displayArray.length; row++)
     {
         System.out.print((row + 1) + "     ");
         for (column=0; column < displayArray[row].length; column++)
             System.out.print("$" + displayArray[row][column] + " ");
         System.out.println();
     }
}
```

InterestTable3是个完整的程序，其中使用了这个方法。该程序可以从本书的Web站点上的源代码中得到。

这里的showTable版本可以处理有任何行数和列数的数组。在图6-18的程序中，如果用这里的版本，其表现将同原来的版本相同，但是这个版本更通用，能够适用于更多不同的情况。

图6-19　重新定义showTable方法

快速参考：多维数组形参

方法的形参可以是整个多维数组。其语法同一维数组形参几乎一样，只是要用更多的方括号[]。

举例（多维数组形参）：

```
public static void showOneElement(char[][] a, int row, int column)
{
     System.out.print(a[row][column]);
}

public static void reinitialize(int[][] anArray)
{
     int row, column;
     for (row = 0; row < anArray.length; row++)
         for (column = 0; column < anArray[row].length; column++)
             anArray[row][column] = 0;
}
```

将在6.5.3节中解释length。

举例（数组实参）：

```
char[][] page = new char[100][80];
int[][] a = new int[10][20];
int[][] b = new int[30][40];
    <这里是某些填充数组的代码。>
showOneElement(page, 5, 10);
reinitialize(a);
reinitialize(b);
```

注意数组a和b的维度不同。还要注意数组实参没有使用方括号。

（前面的例子都位于一个方法的定义之中。所有的方法都假设在同一个类中。）

快速参考：返回多维数组

方法可以将多维数组作为返回值。其语法几乎同使用一维数组作为返回值一样，只是要用更多的方括号[]。

语法：

public static *Base_Type*[]...[] *Method_Name*(*Parameter_List*)
Method_Body

也可以用其他修饰符来代替public static。

举例（假设在某个类的定义中）：

```
public static char[][] blankPage(
                    int numberOfLines, int charPerLine)
{
    char[][] newArray = new char[numberOfLines][charperLine];
    int line, character;
    for (line = 0; line < numberOfLines; line++)
        for (character= 0; character < charPerLine; character++)
            newArray[line][character] = ' ';
    return newArray;
}
```

6.5.4 不规则数组（选读）

既然二维数组在Java中是数组的数组，那就没有必要让每一行有完全相同数目的元素。换种略微不同的表达方式，即不同的行可以有不一样的列数。这种数组叫作**不规则数组**(ragged array)。

下面从一个普通的、规则的二维数组开始，来说明其中的奥妙。这个数组是这样创建的：

```
int[][] a = new int[3][5];
```

这和下面的代码等价：

```
int[][] a;
a = new int[3][];
a[0] = new int[5];
a[1] = new int[5];
a[2] = new int[5];
```

其中，代码行

```
a = new int[3][];
```

使a是一个长度为3的数组的名字，该数组的每个元素是一个基类型为int的数组名，这些数组的长度不限。下面3行各自创建了一个长度为5的基类型为int的数组，分别命名为a[0]、a[1]及a[2]。最后的结果就是一个基类型为int的二维数组，有3行5列。

下面的语句引发了问题“它们的长度必须都是5吗？”答案是否定的。

```
a[0] = new int[5];
a[1] = new int[5];
a[2] = new int[5];
```

下面的代码定义了一个类似的数组b，不过它是不规则的，每行的长度都不相同：

```
int[][] b;
b = new int[3][];
b[0] = new int[5];
```

```
b[1] = new int[7];
b[2] = new int[4];
```

值得注意的是，填充好前面的数组b之后，可以用图6-19中的showTable方法来显示它。然而，图6-18中的showTable方法不能用于显示b。

在有些情况下，使用不规则数组是有好处的，不过绝大多数应用程序并不需要它。即便如此，理解了不规则的数组，就会更好地理解Java中的各种多维数组是如何运作的。

编程示例：雇员出勤时间记录

在这个编程示例中，使用一个名为hours的二维数组来保存公司里每个雇员从周一到周五每天的工作时间。该数组的第一个索引用来指明一周中的具体某天，第二个索引指明是哪个雇员。这个二维数组是图6-20中的TimeBook类的一个私有实例变量。该类的main方法中带有一个演示程序，模拟了一个仅有3个雇员的小公司的情况。雇员的编号从1开始，数组的索引编号是从0开始的，因此描述雇员的数组索引要根据雇员编号减1。例如，编号为3的雇员在星期二的工作时间记录在hours[1][2]中。第一个索引代表了工作日的第二天（星期二），第二个索引代表第三号雇员。对工作日的编号是0代表星期一，1代表星期二，以此类推。雇员的编号是1、2、3但是保存在数组索引中的位置是0、1及2。

图6-20中的TimeBook类还没有全部完成，还需要更多的方法才能使之成为一个有实用价值的类，不过它已经有足够多的方法来支持main中的演示程序了。图6-20中的定义只是实现该类的一个初级阶段，其中的setHours方法甚至还只是一个空架子。这个空架子只是定义了该方法使之能用于测试，但还不是最终的实现。在后面的编程项目中，有个任务就是要把这个类的定义完成。不过，当前来说，它已经足够用于演示二维数组hours的用法了，这个数组是该类的一个实例变量。

```
/**
 Class for a one-week record of the time worked by
 each employee. Uses a five-day week (Mon.-Fri.).
 main has a sample application.
*/
public class TimeBook
{
    private int numberOfEmployees;
    private int[][] hours;
        //hours[i][j] has the hours for employee j on day i.
    private int[] weekHours;
        //weekHours[i] has the week's hours worked for
        //employee i+1.
    private int[] dayHours;
        //dayHours[i] has the total hours worked by all
        //employees on day i. Monday is 0, Tuesday 1, etc.

    /**
     Reads hours worked for each employee on each day of
     the week into the two-dimensional array hours. (The method
     for input is just a stub in this preliminary version.)
     Computes the total weekly hours for each employee and
     the total daily hours for all employees combined.
    */
    public static void main(String[] args)
    {
```

图6-20 出勤时间记录程序

```
        TimeBook book = new TimeBook(3);
        book.setHours();
        book.update();
        book.showTable();
    }

    public TimeBook(int theNumberOfEmployees)
    {
        numberOfEmployees = theNumberOfEmployees;
        hours = new int[5][numberOfEmployees];
        //the 5 is for the 5 days Monday through Friday.
        weekHours = new int[numberOfEmployees];
        dayHours = new int[5];
    }
    public void setHours() //This is just a stub.
    {
        hours[0][0] = 8; hours[0][1] = 0; hours[0][2] = 9;
        hours[1][0] = 8; hours[1][1] = 0; hours[1][2] = 9;
        hours[2][0] = 8; hours[2][1] = 8; hours[2][2] = 8;
        hours[3][0] = 8; hours[3][1] = 8; hours[3][2] = 4;
        hours[4][0] = 8; hours[4][1] = 8; hours[4][2] = 8;
    }

    public void update()
    {
        computeWeekHours();
        computeDayHours();
    }

    private void computeWeekHours()
    {
        int dayNumber, employeeNumber, sum;
        for (employeeNumber = 1;
                employeeNumber <= numberOfEmployees; employeeNumber++)
        {//Process one employee:
            sum = 0;
            for(dayNumber = 0; dayNumber < 5; dayNumber++)
                sum = sum + hours[dayNumber][employeeNumber - 1];
                //sum contains the sum of all the hours worked
                //in one week by employee with number employeeNumber.
            weekHours[employeeNumber - 1] = sum;
        }
    }

    private void computeDayHours()
    {
        int dayNumber, employeeNumber, sum;
        for (dayNumber = 0; dayNumber < 5; dayNumber++)
        {//Process one day (for all employees):
            sum = 0;
            for (employeeNumber = 1;
                            employeeNumber <= numberOfEmployees;
```

实际的类会有更多方法，我们只列出了main中用到的。

图6-20 （续）

```
                                   employeeNumber++)
            sum = sum + hours[dayNumber][employeeNumber - 1];
            //sum contains the sum of all hours worked by all
            //employees on day dayNumber.
        dayHours[dayNumber] = sum;
    }
}
public void showTable()
{
    int row, column;
    System.out.print("Employee   ");
    for (column = 0; column < numberOfEmployees; column++)
        System.out.print((column + 1) + "     ");
    System.out.println("Totals");
    System.out.println();

    for (row = 0; row < 5; row++)
    {
        System.out.print(day(row) + " ");
        for (column = 0; column < hours[row].length; column++)
            System.out.print(hours[row][column] + " ");
        System .out.println(dayHours[row]);
    }
    System.out.println();

    System.out.print("Total = ");
    for (column = 0; column < numberOfEmployees; column++)
        System.out.print(weekHours[column] + "     ");
    System.out.println();
}

//Converts 0 to "Monday", 1 to "Tuesday" etc.
//Blanks used to make all strings the same length.
private String day(int dayNumber)
{
    String dayName = null;
    switch (dayNumber)
    {
        case 0:
            dayName = "Monday   ";
            break;
        case 1:
            dayName = "Tuesday  ";
            break;
        case 2:
            dayName = "Wednesday";
            break;
        case 3:
            dayName = "Thursday ";
            break;
        case 4:
            dayName = "Friday   ";
```

ShowTable方法能够也应该更健壮，参见编程项目6。

图6-20　（续）

```
                break;
            default:
                System.out.println("Fatal Error.");
                System.exit(0);
                break;
        }
        return dayName;
    }
}
```

屏幕对话示例

```
<在最终的程序中，setHours的空架子会被实际的方法替换，用一个输入对话来
采集每个雇员每天的工作时间。>
Employee    1    2   3   Totals
Monday      8    0   9   17
Tuesday     8    0   9   17
Wednesday   8    8   8   24
Thursday    8    8   4   20
Friday      8    8   8   24
Total =     40   24  38
```

图6-20 （续）

除了二维数组hours之外，TimeBook类还用了两个普通的一维数组作为实例变量。数组weekHours用于记录每个雇员在整个星期的总工作小时数。computeWeekHours方法将weekHours[0]设置为等于1号雇员整个星期的总工作小时数，weekHours[1]设置为等于2号雇员整个星期的总工作小时数，依次类推。

数组dayHours用于记录一个星期里每一天中所有雇员工作的总小时数。computeDayHours方法将dayHours[0]设置为等于周一所有的雇员合计的总工作小时数，dayHours[1]设置为等于周二所有雇员的总工作小时数，依次类推。图6-21画出了数组hours、weekHours及dayHours之间的相互关系。在那个图中，显示了数组hours的一些示例数据，这些数据又决

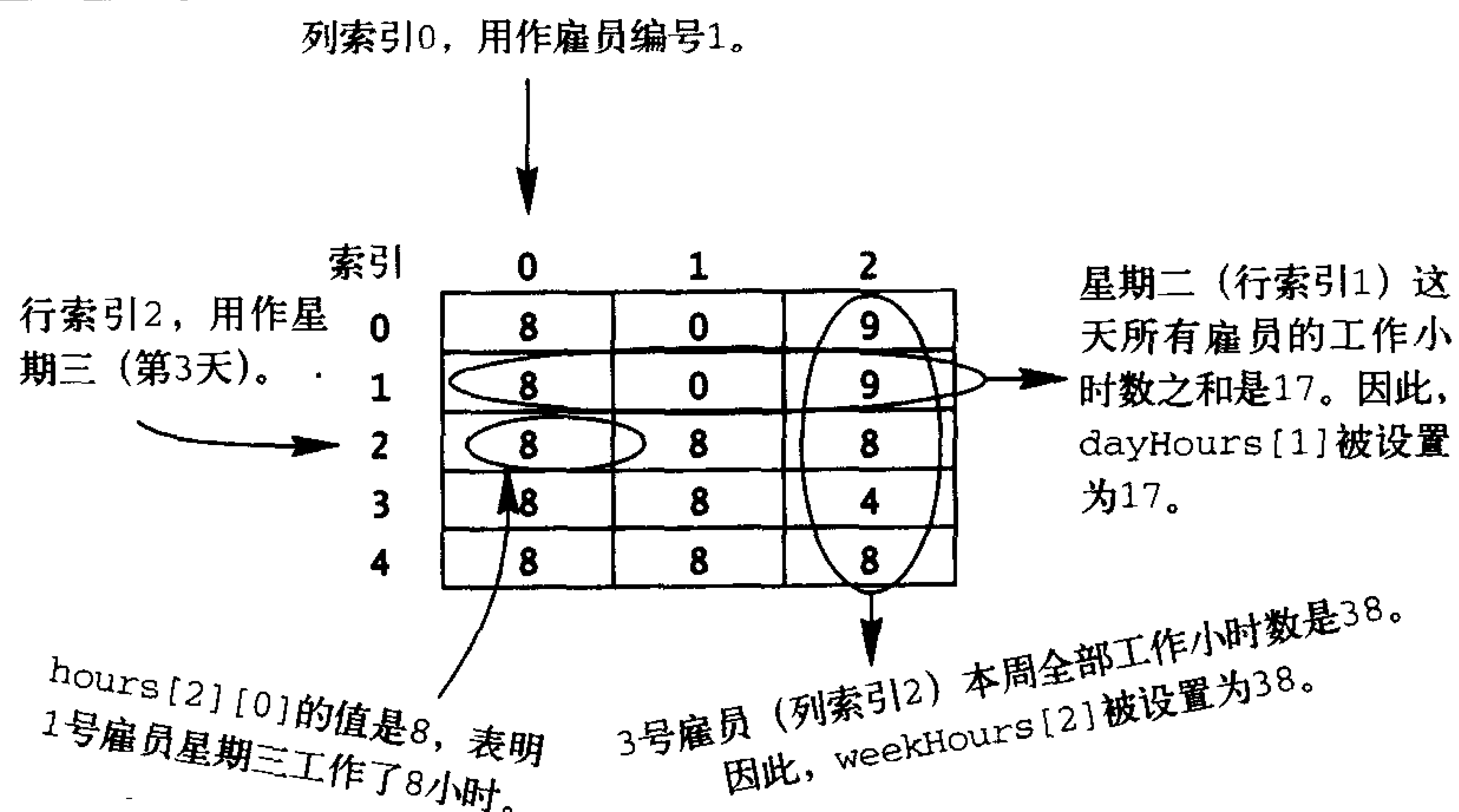


图6-21 TimeBook类中的数组

定了weekHours和dayHours中存放的值。

一定要仔细注意computeWeekHours方法是如何使用二维数组hours的索引的。其中有一个for循环嵌套在另一个for循环之内。外面的for循环遍历了所有的雇员，内部的for循环遍历一周的每一个工作日。那个内部的for循环（连同对变量sum的初始化及其后的一条赋值语句）摘录如下：

```
sum = 0;
for(dayNumber = 0; dayNumber < 5; dayNumber++)
    sum = sum + hours[dayNumber][employeeNumber - 1];
    //sum contains the sum of all the hours worked
    //in one week by the employee with number employeeNumber.
weekHours[employeeNumber - 1] = sum;
```

注意，在计算某个雇员工作小时数之和时，用来代表特定的雇员的第二个索引保持为常量。

computeDayHours方法用类似的方式计算每周的每个工作日里所有雇员的工作小时数总和。不过，在这种情况下，内部的for循环用第二个索引进行遍历，而将第一个索引保持为常量。换句话来说，雇员索引和周内工作日索引的角色对调了。

TimeBook类还算不上是个已完成而能被反复复用的软件片段。现在已经做的都是对的，但是还没有完工。setHours方法只是一个空架子，需要被替换为一个更实际可用的方法，让用户用键盘来输入小时数。除非所有的小时数都有同样多的数字，否则showTable方法不能显示得像图6-20所示那么整洁。因此，showTable方法还需要变得更健壮以便把各种可能的工作小时数都整洁地显示出来。最后，TimeBook类还应当有更多的方法使之能在更广泛的场合下使用。编程项目6中给出的任务就是针对这些方面把TimeBook类加以完善的。

自测题

22. 下列代码会产生什么样的输出？

```
int[][] testArray = new int[4][4];
int index1, index2;
for (index1 = 0; index1 < testArray.length; index1++)
    for (index2 =0; index2 < testArray[index1].length; index2++)
        testArray[index1][index2] = index2;
for (index1 = 0; index1 < testArray.length; index1++)
{
    for (index2 =0; index2 <testArray[index1].length; index2++)
        System.out.print(testArray[index1][index2] + " ");
    System.out.println();
}
```

23. 编写代码，从键盘输入数字来填充下面所声明的数组a：

```
int[][] a = new int[4][5];
```

每行将输入5个数字，共计4行（然而答案不应当依赖于输入的数字是怎样分行的）。

24. 编写一个display方法的方法定义，它不返回值，像下面这样对该方法的调用将把自测题23中的数组a按照对输入所描述的那种格式显示出来，也就是占4行，每行5个数字：

```
display(a);
```

该方法的实现也应当可以用于其他尺寸不是4×5的二维数组。将方法限定为静态的，可以加入某个类之中。

6.6 图形编程补充（选读）

将答案写在所提供的空间中。

——Web页面上的指示

这里的补充材料的第一部分不依赖于本章中前面的任何内容，因此就算没有读过本章的主体部分，也可以阅读。第二部分则需要先阅读本章的主体，以掌握数组的基本知识。

6.6.1节中，讲述了文本区和文本域，它们是可以用在applet中进行文本输入输出的组件。6.2.2节中，讲解如何在applet中画出通用的多边形（polygon）图案。

6.6.1 文本区和文本域

文本区是applet中一个类似窗口的区域，可以用于文本的输入和输出。文本区中可以容纳任意多行，每行可以有任意多个字符。文本域和文本区类似，但它只能容纳一行文本。例如，用户可以在图6-22所示的applet的文本区里提出问题，这个applet最终会把答案显示在同一个文本区中。我们先来讨论图6-22中的applet运行时会发生什么，然后，讨论对图6-22中的文本区及一般的文本区（还有文本域）编程的细节。

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Oracle extends JApplet implements ActionListener
{
    public static int LINES = 5;
    public static int CHAR_PER_LINE = 40;

    private JTextArea theText;
    private String question;
    private String answer;
    private String advice;

    public void init()
    {
        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());

        JLabel instructions =
            new JLabel("I will answer any question, " +
                        "but may need some advice from you.");
        contentPane.add(instructions);

        JButton getAnswerButton = new JButton("Get Answer");
        getAnswerButton.addActionListener(this);
        contentPane.add(getAnswerButton);

        JButton sendAdviceButton = new JButton("Send Advice");
        sendAdviceButton.addActionListener(this);
```

图6-22 有文本区的applet

```
        contentPane.add(sendAdviceButton);

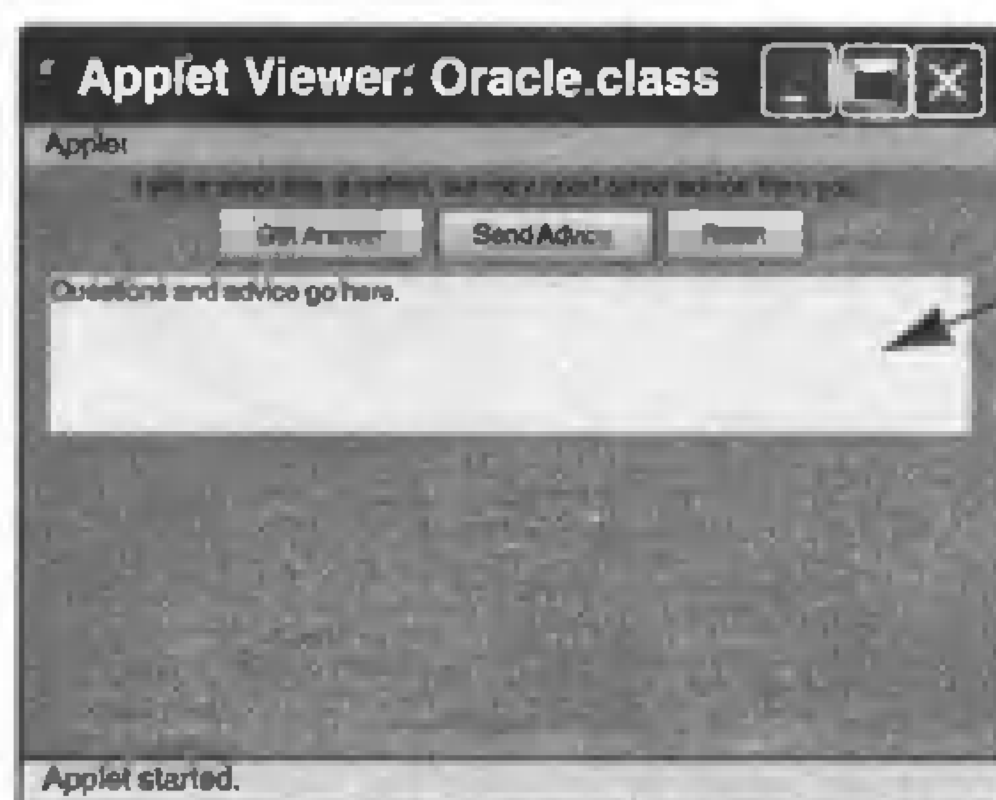
        JButton resetButton = new JButton("Reset");
        resetButton.addActionListener(this);
        contentPane.add(resetButton);

        theText = new JTextArea(LINES, CHAR_PER_LINE);
        theText.setText("Questions and advice go here.");
        contentPane.add(theText);

        answer = "The answer is: Look around."; //for first answer
    }

    public void actionPerformed(ActionEvent e)
    {
        String actionCommand = e.getActionCommand();
        if (actionCommand.equals("Get Answer"))
        {
            question = theText.getText();
            theText.setText("That is a difficult question.\n" +
                        "I need some advice.\n" +
                        "Give me some advice and click button.");
        }
        else if(actionCommand.equals("Send Advice"))
        {
            advice = theText.getText();
            theText.setText("That advice helped.\n" +
                        "You asked the question: " + question
                        + "\n" + answer +
                        "\nClick the Reset button and" +
                        "\nLeave the program on for others.");
            answer = "The answer is: " + advice;
        }
        else if(actionCommand.equals("Reset"))
        {
            theText.setText("Questions and advice go here.");
        }
        else
            theText.setText("Error");
    }
}
```

得到的applet



这是文本区。

按钮的位置取决于用户计算机的本地条件。要清除显示，可以用鼠标改变applet窗口的大小。

图6-22　(续)

编程示例：问一答applet

图6-22中的applet专门回答问题，不管是什么样的问题都可以。用鼠标把文本区中的文字激活，然后输入任何问题。此时，文本区中除了有这个问题之外什么都没有。要获得答案，点击Get Answer按钮。不过，点击Get Answer按钮没有产生出答案，而是出现一些文本请用户为寻找答案提供建议。这些请求建议的文本就在文本区中。和输入问题一样，用户在文本域中输入一些建议，然后点击Send-Advice按钮。这最终导致applet在文本区中给出了答案。阅读答案之后，点击Reset按钮，使applet返回到初始状态。假定用户会让applet继续运行，直到另一个用户来问另一个问题。这样就可以让任意多的用户来提问、给出建议以及获得答案。

假设有一些用户需要提问题。如果每个用户只允许提出一个问题，在此之后必须到其他等待提问的用户后面去排队，这样这个applet看起来就很机智。不过，若站在一边看这些用户，很快就能意识到一个用户输入的建议被它当作答案给了下一个用户。这不是个很有想象力的花招，却是一个使用文本域进行输入输出的好例子。6.6.2节中将讲述图6-22中的文本区的细节以及一般的文本域。

JTextArea和JTextField对象

文本区（text area）是JTextArea类的对象，显示为一个允许用户输入多行文本的域。图6-22中的下列代码创建了一个名为theText的文本区，用户在其中提问以及获得答案：

```
private JTextArea theText;
        ...
theText = new JTextArea(LINES, CHAR_PER_LINE);
```

在图6-22中，变量theText是私有的实例变量。前面的最后一行创建JTextArea的代码在applet的init方法中。传递给JTextArea类的构造函数的实参是预定义的常量LINES和CHAR_PER_LINE，用于指定这个文本区将有空间容纳至少LINES行每行至少CHAR_PER_LINE个字符。LINES和CHAR_PER_LINE定义为5和40，因此这个文本区将能容纳5行，每行40个字符。用户可以在这个文本区中输入任意多文本，但是只有有限的一部分能显示出来。在这个例子中，至少5行每行至少40个字符可以显示出来。

applet可以读取文本区中的文本以得到文本输入，还能够让文本显示在文本区中以进行输出。getText方法返回文本区中写入的文本。例如，下面摘自图6-22所示的代码调用getText方法时文本区theText中的字符串赋值给名为question的字符串变量：

```
question = theText.getText();
```

getText方法是输入方法，setText方法是输出方法。setText方法用于在文本区中显示新的文本。例如，下面摘自图6-22所示的代码将使文本区theText改变它显示的文本：

```
theText.setText("That is a difficult question.\n" +
                  "I need some advice.\n" +
                  "Give me some advice and click button.");
```

则文本区中显示的文本将换为下面的字符串：

```
"That is a difficult question.\n" +
"I need some advice.\n" +
"Give me some advice and click button."
```

注意这个字符串中有两个断行，因此会被显示为3行文本。

文本域是JTextField类的对象。创建和使用文本域的细节同文本区一样，只是文本域只能容纳一行文本，并且使用了JTextField类而不是JTextArea类。

快速参考：JTextArea类和JTextField类

JTextArea类和JTextField类可以向图形界面增添允许被修改的文本。JTextArea类的对象有

一个指定能容纳多少行以及每行多少个字符的尺寸。JTextField类的对象只能有一行，可容纳指定数目的字符。可以输入比指定的尺寸更多的文本到JTextArea类或JTextField类的对象中，但是额外的文本不一定能显示出来。

举例（使用JTextArea）：

```
JTextArea someText = new JTextArea(10, 30);
getContentPane().add(someText);
```

举例（使用JTextField）：

```
JTextField name = new JTextField(30);
getContentPane().add(name);
```

快速参考：getText和setText

JTextArea类和JTextField类都有名为getText和setText的方法。getText方法用来获得文本区或文本域中写入的文本。setText方法用来改写文本区或文本域中的文本。

举例（theText是JTextArea类或者JTextField类的对象）：

```
String question = theText.getText();
theText.setText("The answer is 42.");
```

自测题

25. 假设theText是JTextArea类的对象。编写一行代码来改变theText中的内容，使新的内容是由以前的内容写两遍得到。

6.6.2　画多边形

多边形（名词）由三条或更多线段围成的封闭的平面图形。

——《美国传统英语词典》，（第3版）

本小节不依赖于6.6.1节，不过却依赖于本章主体中讲述的数组基本知识。本节讲述了Graphics类中的一些方法，可以用来画一类通用的多边形图案。

画多边形

Graphics类中的drawRect方法可以在applet中画矩形。矩形总有90°（直角）的角，然而，有时候需要其他角度的图形。drawPolygon方法可以用来画出任意多边形。多边形是由不交叉的线段组成的封闭图形。矩形是多边形的特例，多边形可以有任意多条边而且边之间的角度不必是90°。可以通过给出顶点位置的方法来确定多边形。多边形的图形可以从画出连接相继的顶点的线段来得到。例如，图6-23中显示了顶点以及连接顶点的线段以构成多边形。顶点p1、p2、p3、p4、p5再回到p1就构成了多边形。用drawPolygon方法画出这个多边形，需要3个参数：第一个是int数组，给出顶点的x坐标，第二个是另一个int数组，给出顶点的y坐标，第三个是顶点数。使用drawPolygon方法时，那两个数组有相同的长度（大小），而且第三个参数是数组的长度。

图6-24展示了一个applet，它用到了drawPolygon以及另外两个紧密相关的方法。这个applet显示了一个屋子的简单图形。窗户是用drawPolygon方法画的，屋子本身是用fillPolygon方法画的。fillPolygon方法所用的实参和drawPolygon方法相同，但是

fillPolygon方法能够把多边形内部填充为彩色。

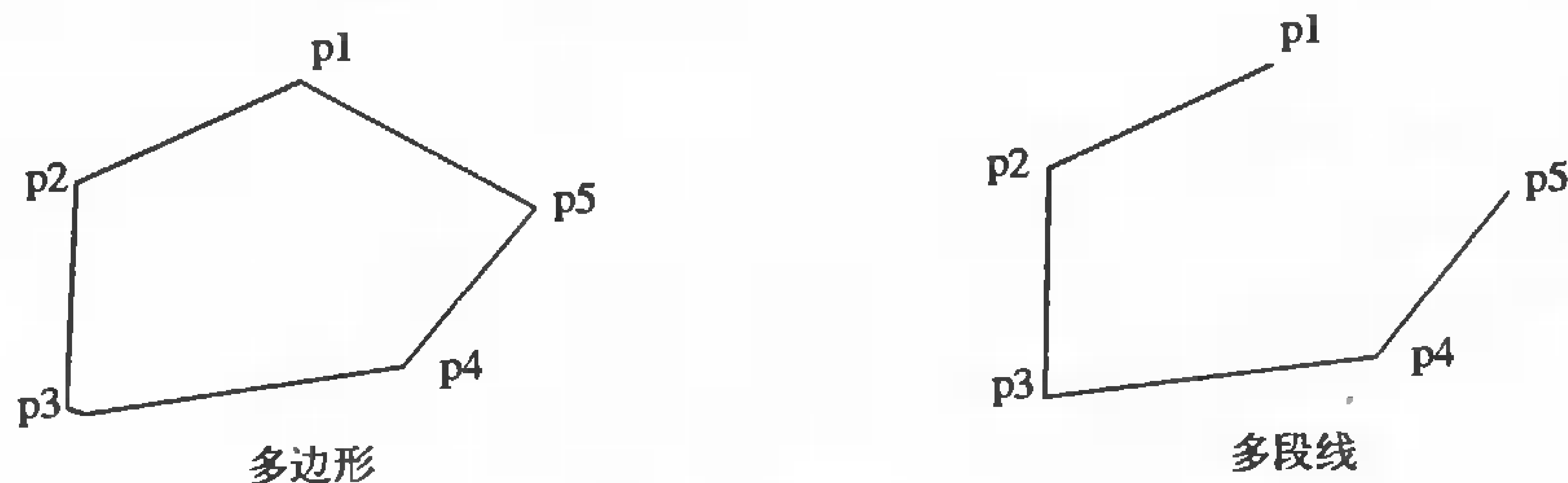


图6-23 多边形和多段线

多段线（polyline）和多边形类似，只是不需要封闭。图6-23中显示了多边形和多段线的不同之处。对于多段线来说，没有从最后顶点到初始顶点的那条边。Graphics类的方法drawPolyline可以画多段线。图6-24中的applet用drawPolyline方法画屋子的门。

```
import javax.swing.*;
import java.awt.*;

public class House extends JApplet
{
    private int[] xHouse = {150, 150, 200, 250, 250};
    private int[] yHouse = {100, 40, 20, 40, 100};
    private int[] xDoor = {175, 175, 200, 200};
    private int[] yDoor = {100, 60, 60, 100};
    private int[] xWindow = {220, 220, 240, 240};
    private int[] yWindow = {60, 80, 80, 60};

    public void paint(Graphics canvas)
    {
        this.setBackground(Color.LIGHT_GRAY);
        canvas.setColor(Color.GREEN);
        canvas.fillPolygon(xHouse, yHouse, xHouse.length);
        canvas.setColor(Color.BLACK);
        canvas.drawPolyline(xDoor, yDoor, xDoor.length);
        canvas.drawPolygon(xWindow, yWindow, xWindow.length);
        canvas.drawString("Home sweet home!", 150, 120);
    }
}
```

得到的applet

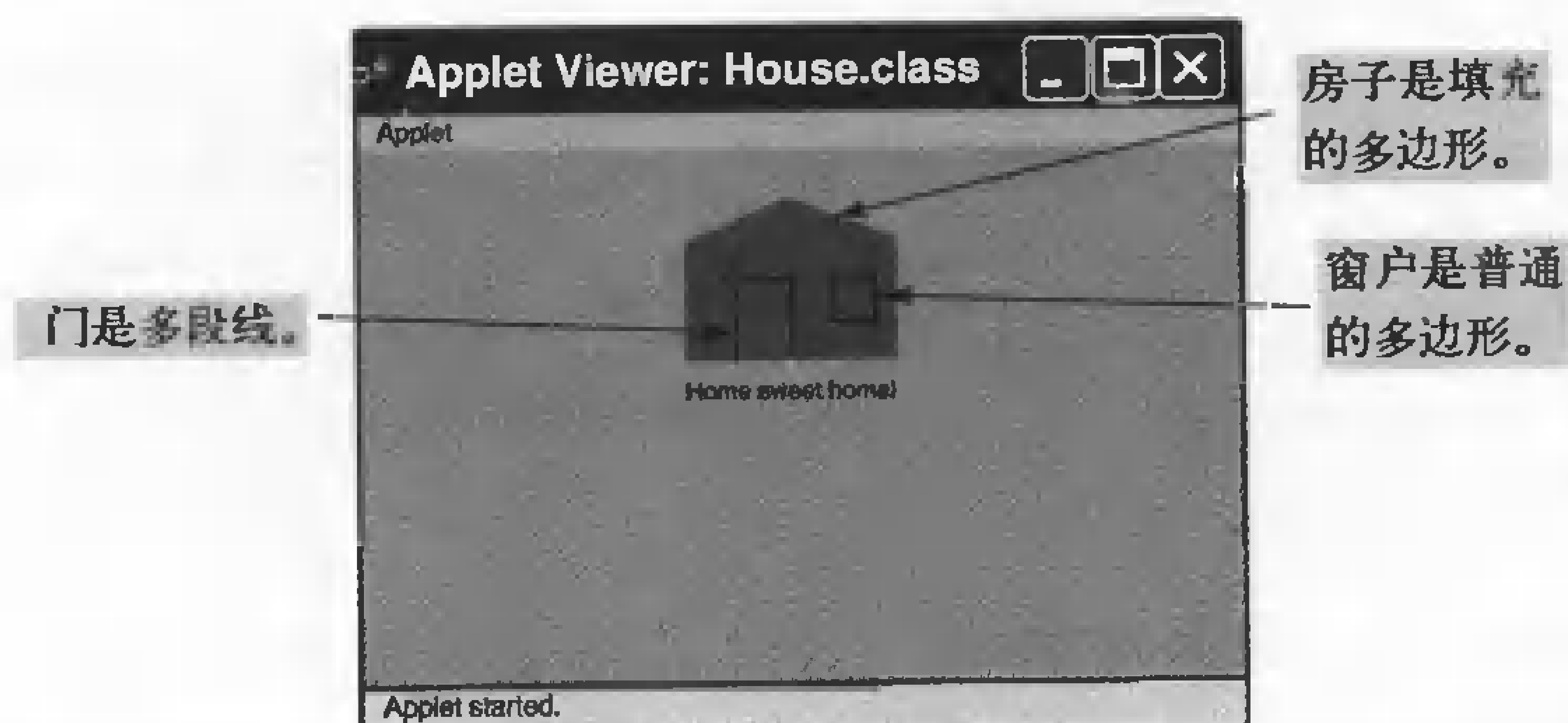


图6-24 有多边形和多段线的applet

自测题

26. drawRect方法可以画矩形，但drawPolygon方法也能画矩形。以canvas作为调用对象，写出对drawPolygon方法的调用，使之画一个矩形，位于点（7，8），高10个像素，宽20个像素。

快速参考：drawPolygon、fillPolygon及drawPolyline

drawPolygon、fillPolygon及drawPolyline都是Graphics类的方法。

drawPolygon：该方法用于画多边形。

语法：

canvas.drawPolygon(*Array_Of_xs*, *Array_Of_ys*, *Number_ Of_Points*);

用下列连接顶点的线段画出多边形：

(*Array_Of_xs*[0], *Array_Of_ys*[0]), (*Array_Of_xs*[1], *Array_Of_ys*[1])...

...(*Array_Of_xs* [*Number_Of_Points*], *Array_Of _ys*[*Number_Of_Points*])

举例：

```
private int[] xCoord = {150, 150, 200, 250, 250};
private int[] yCoord = {100, 40, 20, 40, 100};
        ...
canvas.drawPolygon(xCoord, yCoord, xCoord.length);
```

fillPolygon：该方法和drawPolygon一样画多边形，但是对多边形的内部进行填充。

语法：

canvas.fillPolygon(*Array_Of_xs*, *Array_Of_ys*, *Number_Of_Points*);

举例：

```
canvas.fillPolygon(xCoord, yCoord, xCoord.length);
```

drawPolyline：该方法画的形状和drawPolygon方法类似，只是从最后一个顶点到第一个顶点之间没有线段，因此通常图形不是封闭的。

语法：

canvas.drawPolyline(*Array_Of_xs*, *Array_Of_ys*, *Nunber_Of_Points*);

举例：

```
canvas.drawPolyline(xCoord, yCoord, xCoord.length);
```

小　结

- 数组可以被看作相同类型变量的集合。
- 数组是用new创建的对象，就像本章以前讨论过的类对象一样（尽管它们所用的语法略有不同）。
- 数组中索引变量从0开始编号，到数组的长度减1为止。若a是个数组，则a[i]是数组a的一个索引变量。索引i必须是一个大于或等于0而且严格小于a.length的值。若i是其他值，这种情况就被称为**数组索引越界错误**（array index out-of-bounds error），在运行程序时将得到错误信息。
- 当索引变量被用作方法的实参时，就和与其基类型相同的其他实参被同样对待。尤其是，若基类型是基本类型，方法就不能修改索引变量的值；而若基类型是类，方法就可以修改位于索引变量的元素。
- 方法可以将数组作为其返回值。
- 当仅使用数组的一部分时，通常把值保存在数组的起始部分，并用一个int类型的变量记录在数组

中保存了多少值。这样的数组叫作部分填充的数组。

- 当访问方法返回一个私有的数组类型的实例变量时，要小心处理，应当返回该数组的一份副本，而不是返回该私有实例变量本身。
- 可以用选择排序算法对一系列值进行排序，比如把数值按照递增或者递减的顺序排序。
- 数组可以有不止一个索引，这就是多维数组。
- 二维数组可以被当作二维的表格，第一个索引是行，第二个索引是列。
- 多维数组在Java中的实现是数组的数组。
- 可以在applet中加入文本域和文本区，让applet能够进行文本输入和输出。
- 可以在applet中画任意多边形。①

✓ 自测题答案

1.
```
0 2 4 6 8 10 12 14 16 18
```

2.
```
a
e
i
o
u
```

3.
```
Tide 1 is -7.3
Tide 2 is 14.2
```

4. for循环引用了元素b[1]到b[10]，但是索引为10的元素不存在。数组的元素是b[0]到b[9]。若这段代码包含在一个完整的类或程序中，它可以通过编译，不会出错误信息；但是若运行，就会得到错误信息，指出数组的索引越界了。

5. a的最后一个索引是9。a.length的值是10。

6.
```
import java.util.*;

public class Exercise
{
    public static void main(String[] args)
    {
        double[] a = new double[20];

        int index;
        Scanner keyboard = new Scanner(System.in);
        System.out.println("Enter 20 numbers:");
        for (index = 0; index < a.length; index++)
            a[index] = keyboard.nextDouble();

        System.out.println(
            "The numbers and differences from last number are:");
        for (index = 0; index < a.length; index++)
            System.out.println(a[index]
                + " differs from last by "
                + (a[a.length - 1] - a[index]));
    }
}
```

7.
```
SalesAssociate[] entry = new SalesAssociate[3];
```

① 6.6节中介绍的内容。

```
    int i;
    for (i = 0; i < entry.length; i++)
        entry[i] = new SalesAssociate("Jane Doe", 5000);
```

8. 修改过的代码行已经用**颜色**在下面标出。注意，还应当修改SalesAssociate类的writeOutput方法的定义。该方法的代码下面也已经列出。记住必须让Dollars类可用，可以将其放在同一个目录中，或者将其打包，并导入那个包。

```
/**
 Displays sales report on console screen.
*/
public void displayResults()
{
    System.out.println("Average sales per associate is $" + average);
    System.out.println("The highest sales figure is $" + highest);
    System.out.println();
    int i;
    System.out.println("The following had the highest sales:");
    for (i = 0; i < numberOfAssociates; i++)
    {
        double nextSales = record[;].getSales();
        if (nextSales == highest)
        {
            record[i].writeOutput();
            Dollars.write(nextSales - average);
            System.out.println(" above the average.");
            System.out.println();
        }
    }

    System.out.println("The rest performed as follows:");
    for (i = 0; i < numberOfAssociates; i++)
    {
        double nextSales = record[i].getSales();
        if (record[i].getSales() != highest)
        {
            record[i].writeOutput();
            if (nextSales >= average)
            {
                Dollars.write(nextSales - average);
                System.out.println(" above the average.");
            }
            else
            {
                Dollars.write(average - nextSales);
                System.out.println(" below the average.");
            }
            System.out.println();
        }
    }
}
```

下面是SalesAssociate类的writeOutput方法的修订版本，改动之处用**颜色**标明：

```
public void writeOutput()
{
```

```
    System.out.println("Sales associates: " + name);
    System.out.print("Sales: ");
    Dollars.writeln(sales);
}
```

9.
```
a[1] = a b[1] = a
a[2] = a b[2] = a
a[1] = a b[1] = a
a[2] = b b[2] = b
```

10.
```
public static void showArray(char[] a)
{
      int i;
      for (i = 0; i < a.length; i++)
          System.out.print(a[i]);
      System.out.println();//This line is optional.
}
```

11.
```
public static double[] halfArray(double[] a)
{
     double[] temp = new double[a.length];
     int i;
     for (i = 0; i < a.length; i++)
         temp[i] = a[i]/2.0;
     return temp;
}
```

12. 若b是个长度为10的数组，下面的调用：

```
doubleSize(b);
```

运行不会出错，但是b的长度不会改变。实际上，b什么都不会变。参数a是个局部变量并被初始化为对b的引用。局部变量a被改变，使其是对一个大小是b两倍的数组的引用，然而该引用当调用结束的时候就无效了。

13.
```
int i;
for (i = 0; i < a.length; i++)
    System.out.println(a[i]);
```

14.
```
int i;
for (i = 0; i < numberUsed; i++)
     System.out.println(a[i]);
```

15. a[0],a[1],and a[2]。

16.
```
a[numberUsed] = 42;
numberUsed++;
```

17.
```
if (numberUsed == a.length)
    System.out.println("List is full. Cannot add 42.");
else
{
    a[numberUsed] = 42;
    numberUsed++;
}
```

18.
```
SelectionSort.sort(myArray);
```

19. 只需要把数组元素的类型改为double。可以简单地把int都替换为double，不过用来指明索引的数据类型的int不能换。例如，可以把

```
private static void interchange(int i, int j, int[] a)
```

替换为

```
private static void interchange(int i, int j, double[] a)
```

注意，i和j都是索引，所以它们还是int类型。

20. 要让代码按照降序排序，只要将下面的indexOfSmallest的定义中的 <替换为>就可以了：

```
if (a[index] < min)
```

不过，为了使代码能被读懂，还应当把该方法的名字indexOfSmallest改为indexOfLargest，把变量名min改为max，把变量名indexOfMin改为indexOfMax，等等。相关的注释也应当修改。

21. 若基类型是int的数组中有某个值出现了两次，用SelectionSort.sort方法对其排序时，排序完成后，数组中会有那个重复的值的两个副本。

22.
```
0 1 2 3
0 1 2 3
0 1 2 3
0 1 2 3
```

23.
```
int[][] a = new int [4][5];
int row, column;

System.out.println("Enter numbers:");
Scanner keyboard = new Scanner(System.in);
for (row = 0; row < 4; row++)
     for (column = 0; column < 5; column++)
         a[row][column] = keyboard.nextInt();
```

或者也可以如下所示：

```
System.out.println("Enter numbers:");
Scanner keyboard = new Scanner(System.in);
for (row = 0; row < a.length; row++)
    for (column = 0; column < a[row].length; column++)
        a[row][column] = keyboard.nextInt();
```

24.
```
public static void display(int[][] anArray)
{
    int row, column;
    for (row = 0; row < anArray.length; row++)
    {
        for (column = 0; column < anArray[row].length;
                                              column++)
            System.out.print(anArray[row][column] + " ");
        System.out.println();
    }
}
```

25.
```
theText.setText(theText.getText() + theText.getText());
```

26.
```
private int[] xCoord = {7, 27, 27, 7};
private int[] yCoord = {8, 8, 18, 18};
                ...
canvas.drawPolygon(xCoord, yCoord, xCoord.length);
```

● 编程项目

1. 编写一个程序，读入一系列int值，它们每个值占一行。程序要输出这些值的总和，以及读入的每一

个数，并附注说明各自对总和所作贡献的百分比。该程序要问用户有多少个整数，创建相应长度的数组，用输入的整数填充此数组。一个可能的对话过程如下：

```
How many numbers will you enter?
4
Enter 4 integers, one per line:
2
1
1
2
The sum is 6.
The numbers are:
2 33.3333% of the sum.
1 16.6666% of the sum.
1 16.6666% of the sum.
2 33.3333% of the sum.
```

用一个方法，以整个数组作为一个实参，并返回该数组中数值的总和。

2. 编写一个程序，读入一行文本，输出所读入的文本中出现的字母列表，以及每个字母出现的次数。输入行以句号结束，充作标记值。输出字母时应当按照字母表的顺序输出。采用一个长为26，基类型为int的数组，每个索引变量含有相应的字母出现的次数。数组的0号索引变量含有a出现的次数，数组的1号索引变量含有b出现的次数，依次类推。输入中允许大小写字母，但是把大小写不同的字符看作是相等的。（提示：可以使用第5章中讲述的封装类Character中的方法toUpperCase或者toLowerCase。定义一个辅助方法会很有帮助，该方法以一个字符作为参数，返回一个int类型的值，表示该字符的正确的索引，比如'a'返回0，'b'返回1，依次类推。注意可以用一个强制类型转换把一个char类型的值转换为int类型，比如(int)letter。当然，这样不会得到需要的数字，但是如果把这个值减去(int)'a'，就可以得到正确的索引了。允许用户反复输入直到用户指出已经完成为止）。
3. **回文**（palindrome）是指从前向后读和从后向前读都是一样的字符串，比如"warts n straw"或者"radar"。编写一个程序，读入以句号结束的一串字符，判断该字符串（不包括那个句号）是否是一个回文。可以假设输入仅含有字母和空白符号，还可以假定输入的词不超过80个字符。判断是否是回文时，忽略空白符号，并不区分字母的大小写，比如说下面的字符串会被程序判定为回文：

   ```
   "Able was I ere I saw Elba"
   ```

   所编写的程序不必检查输入的字符串是否是正确的英文词语或短语，字符串"xyzczyx"会被该程序判定为回文。程序中带有一个循环，允许用户检查其他字符串，直到请求程序结束为止。为此，应当定义一个静态方法，名为palindrome，开头如下所示：

   ```
   /**
    Precondition: The array a contains letters and
    blanks in positions a[0] through a[used - 1].
    Returns true if the string is a palindrome and
    false otherwise.
   */
   public static boolean palindrome(char[] a, int used)
   ```

   该程序要把输入的字符串读入一个基类型为char的数组中，以该数组和另一个int类型的变量调用上述方法，int类型的变量记录了数组中有多少个字符有效，如6.3.1节中所介绍的。
4. 设计一个名为BubbleSort的类，类似图6-13中给出的SelectionSort类。BubbleSort类的使用方法和SelectionSort完全一样，但是它使用冒泡排序算法。

冒泡排序算法从数组开头到结尾逐个检查相邻的元素对，若它们顺序不对，就进行交换。这样就把数组变得更接近排好序的状态。该过程不断重复直到数组完全排好序。该算法的伪代码如下。

冒泡排序算法对数组a排序

```
重复下列过程直到数组被完全排好序:
for (index = 0; index < a.length - 1; index++)
    if (a[index] > a[index + 1])
        交换a[index]和a[index + 1]的值。
```

冒泡排序算法在数组已经差不多排好序的情况下，效率很高。在其他大多数情况下，该算法的效率无法和别的排序方法相比。

5. 设计一个名为InsertionSort的类，类似图6-13中给出的SelectionSort类。InsertionSort类的使用方法和SelectionSort完全一样，但是它使用插入排序算法。

插入排序算法使用另外一个数组，将被排序的数组中的元素复制到新数组中。复制每个元素时，会按照顺序插在新数组中的正确的位置。这样通常需要在新数组中移动一系列元素。该算法的伪码如下：

插入排序算法对数组a排序

```
for (index = 0; index < a.length; index++)
    把a[index]的值插入数组temp使得
        已经复制进数组temp的元素是排好序的。
把所有元素从temp复制回a。
```

数组temp是sort方法的局部变量。该数组还是部分填充的数组。因此，所有已经复制进去的值都在数组temp的起始部分。

6. 图6-20中的TimeBook类并没有完成。按照正文中的描述把这个类的定义补充完整。尤其是要添加默认的构造函数和访问方法来恢复和修改每个实例变量、每个实例数组变量中的每个索引变量。别忘了把仅有一个框架的setHours改造为一个从键盘获取值的方法。还要定义一个私有方法，该方法有两个int类型的形参，它输出第一个int类型的形参，所占用的字符空间由第二个形参给出，没有被第一个int类型形参的输出字符串占用的位置用空白字符填充。这样做的结果是：例如，把数组中每个被索引的元素恰好输出占4个字符空间（或者任何指定数量的字符空间），因此能够把数组元素输出为整洁的矩形形状。要确保图6-20中的main方法能和这些新的方法正确地协同工作，还有一点，需要编写独立的测试程序来测试这些新的方法。（提示：为了把一个int类型的值n输出为占固定个数的字符空间，可以用Integer.toString(n)把数字转化为字符串值，然后对字符串进行处理。在5.2.4节中已有讨论。）

7. 编写一个名为TicTacToe的类的定义。一个TicTacToe类的对象就是一个TicTacToe游戏。把游戏棋盘保存在一个基类型为char的二维数组里，有3行3列。该类有下列一些方法，能够增添一步棋着、显示棋盘、判断该谁走（X或O）、判断是否有一方获胜、显示获胜者，以及重新初始化游戏为起始状态。为这个类写出main方法，允许用户用键盘和终端玩这个游戏。游戏双方都坐在键盘前面，轮流输入各自的棋着。

8. （本项目需要掌握6.6.2节）编写一个applet，能够显示一个松树的图形，画法是在一个代表树干的小矩形顶部向上画一个三角形，形成树枝。树应当是绿色的，树干是灰色的。

第7章

继　　承

有其父必有其子。

——*谚语*

本章讲述了继承，它是面向对象程序设计中的核心概念之一。通过继承才能使用Java程序设计语言的许多库。继承使得新的类能通过使用已有的类来定义，因而更加容易复用软件。

目标

- 熟悉继承的一般知识。
- 学习如何在Java中定义并使用派生类。
- 学习动态绑定和多态的一般知识及其在Java中的特点。
- 作为选读，学习`JFrame`类，它用于产生从普通Java应用程序而不是Web页面或applet查看程序中启动的视窗界面。

预备知识

为了弄懂本章内容，必须先读第1章～第5章。第6章在这里不需要。

7.1　继承简介

人都有一死，
苏格拉底是人，
因此苏格拉底也不免一死。

——*典型的逻辑三段论*

继承（inheritance）允许定义非常通用的类，然后通过在之前更通用的类的基础上简单地增加新的细节来定义更专用的类。这节约了劳力，因为更专用的类继承了通用类的全部性质，程序员只需要为新的特性编写程序。

例如，可能针对交通工具定义一个类，它的实例变量记录了该交通工具的轮子数目以及最大载客量。然后可能要为汽车定义一个类，该类要继承交通工具类的全部实例变量和方法。汽车类可能还要增添一些实例变量，如油箱中的燃油量、牌照号码，还有一些额外的方法（有些交通工具，比如马和马车，没有油箱，通常也不需要牌照，但是汽车是一种需要这些"额外"东西的交通工具）。如果用了Java的继承机制，汽车类就可以从交通工具类中自动得到其实例变量和方法，只需要描述新增的实例变量和方法即可。

在构造一个Java中的继承示例之前，先要做些准备工作，首先从下面的编程示例开始。

编程示例：Person类

图7-1中有一个简单的Person类。该类确实很简单，只提供了一个属性——人名。这个类本身没有多少直接用处，主要用它来定义其他类。

Person类的大多数方法都很直接，例如，sameName方法和equals方法类似，不过需要注意的是在比较名字时，不区分大写和小写字母。

```
public class Person
{
    private String name;

    public Person()
    {
        name = "No name yet.";
    }

    public Person(String initialName)
    {
        name = initialName;
    }

    public void setName(String newName)
    {
        name = newName;
    }

    public String getName()
    {
        return name;
    }

    public void writeOutput()
    {
        System.out.println("Name: " + name);
    }

    public boolean sameName(Person otherPerson)
    {
        return (this.name.equalsIgnoreCase(otherPerson.name));
    }
}
```

图7-1 基类

7.1.1 派生类

假设要设计一个学院记录管理程序，来保存学生、教员及非教职雇员的记录。这些记录类型有天然的层次式分组：它们都是关于人的记录，学生是人的一个子类，另一个子类是雇员，包括教员和非教职的雇员。学生还分为两个更小的子类：本科生和研究生。这些子类可能还会被进一步分割为更小的子类。

图7-2绘制了这种层次关系的一部分。尽管程序中不一定需要和人或者雇员有关的类，

但用这些类来思考是有用的。例如，每人都有名字，初始化、输出及改变名字的方法对学生、教员和非教职的雇员都是一样的。在Java里，可以定义一个Person类，其中含有一些实例变量，用来表达对人的所有子类（比如学生、教员和非教职的雇员）都适用的属性。该类的定义中也可以有方法来维护这些实例变量。实际上，在图7-1中已经定义了这样的Person类。

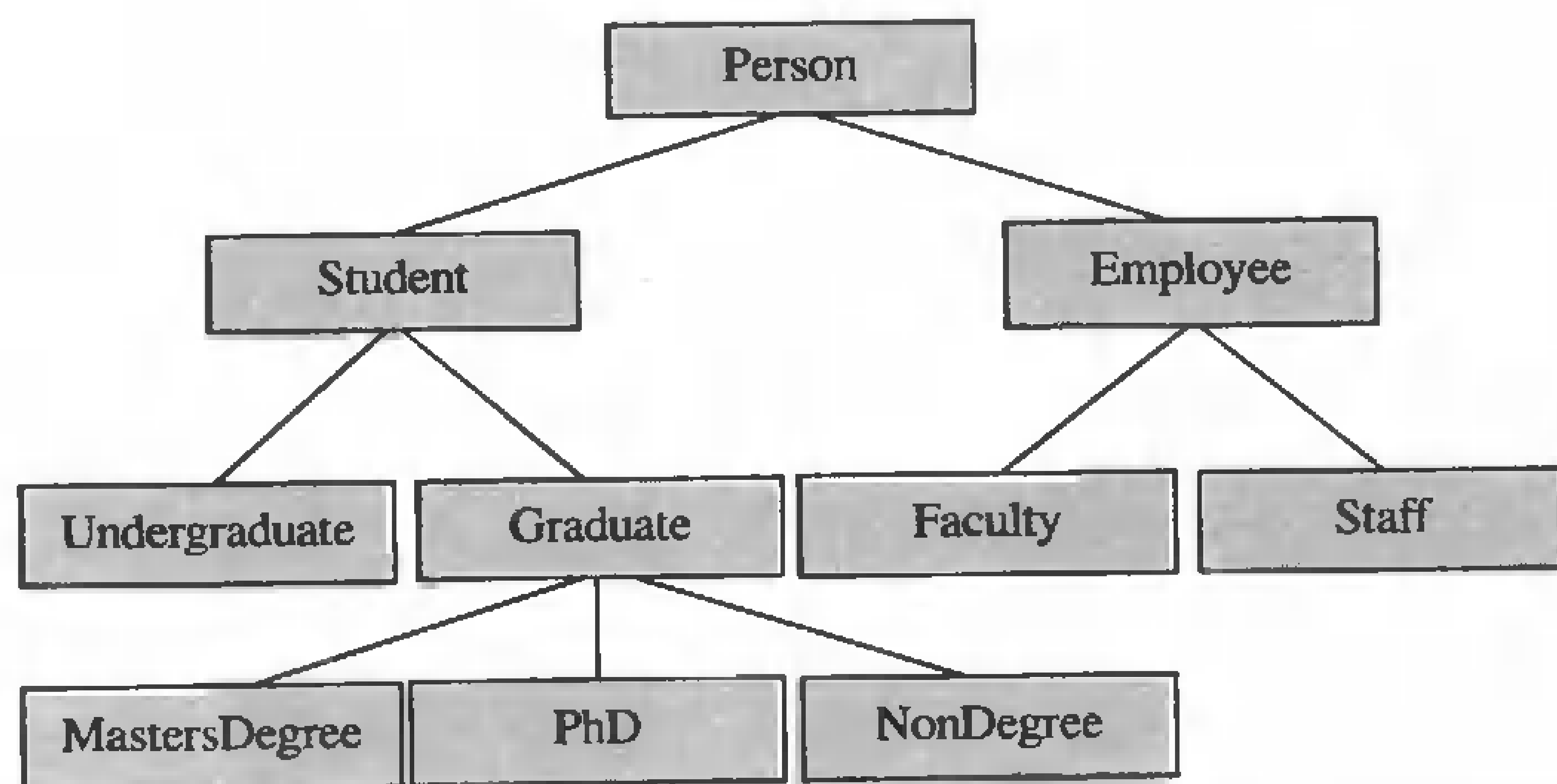


图7-2 类层次图

图7-3是学生类的定义。学生是人，所以我们将Student类定义为Person类的**派生类**(**derived class**)。派生类是通过向一个已经存在的类增添实例变量和方法来定义的。派生类所依赖的已经存在的类称为基类。在这里，Person是基类，Student是派生类。从图7-3中可以看出Student类是Person类的一个派生类：该类的定义的第一行中有extends Person，如下所示开始其定义：

```
public class Student extends Person
```

在定义派生类时，只需要给出添加的实例变量和方法即可。例如，Student类有Person类中的全部实例变量和方法，但在Student类的定义中对它们没有提及。Student类的每个对象都有一个实例变量叫作name，但在Student类的定义中没有说明实例变量name。Student类（或任何派生类）**继承**（**inherit**）了它所扩展的基类的实例变量和方法。

假设用如下的代码创建一个Student类的新对象：

```
Student s = new Student();
```

这里就有一个实例变量s.name。因为name是私有的实例变量，在Person类的定义之外写s.name是不合法的。不过这个实例变量就在那里，可以进行访问及修改。例如，可以用继承的公有的setName方法来修改s.name，代码如下：

```
s.setName("Warren Peace");
```

Student类继承了setName及Person基类的其他方法。

像Student这样的派生类也可以增加一些实例变量和方法。例如，Student类添加了实例变量studentNumber，还有reset、getStudentNumber、setStudentNumber、writeOutput及equals这些方法，还有一些构造方法。（在解释清楚这些类定义的其他部分之后，我们再讨论这些构造方法。）

图7-4中有个很小的演示程序来说明继承。注意，Student类的对象可以调用setName方法，尽管它是其基类Person的方法。Student类从Person类继承了setName。

```
public class Student extends Person
{
    private int studentNumber;

    public Student()
    {
        super();
        studentNumber = 0; //Indicating no number yet
    }

    public Student(String initialName, int initialStudentNumber)
    {
        super(initialName);
        studentNumber = initialStudentNumber;
    }

    public void reset(String newName, int newStudentNumber)
    {
        setName(newName);
        studentNumber = newStudentNumber;
    }

    public int getStudentNumber()
    {
        return studentNumber;
    }

    public void setStudentNumber(int newStudentNumber)
    {
        studentNumber = newStudentNumber;
    }

    public void writeOutput()
    {
        System.out.println("Name: " + getName());
        System.out.println("Student Number: " + studentNumber);
    }

    public boolean equals(Student otherStudent)
    {
        return (this. sameName(otherStudent)
            && (this.studentNumber == otherStudent.studentNumber));
    }
}
```

super在后面的节讲解，在读到正文中对它的讨论之前，不必为之担心。

图7-3 派生类

快速参考：派生类

以一个已经存在的类为基础，通过添加（或改变）方法以及实例变量，能够定义**派生类**。作为基础的类叫做**基类**。派生类从基类中继承了所有实例变量及方法，并可以添加实例变量和方法。

语法：

public class *Derived_Class_Name* extends *Base_Class_Name*

```
{
    Declarations_of_Added_Instance_Variables
    Definitions_of_Added_And_Overridden_Methods
}
```

举例：

参见图7-3。

```
public class InheritanceDemo
{
    public static void main(String[] args)
    {
        Student s = new Student();

        s.setName("Warren Peace");   <-- setName继承自Person类。
        s.setStudentNumber(1234);
        s.writeOutput();
    }
}
```

屏幕输出

```
Name: Warren Peace
Student Number: 1234
```

图7-4 演示继承

7.1.2 重写方法定义

在Student类的定义中，我们添加了一个没有参数的writeOutput方法（见图7-3）。可是Person类也有个writeOutput方法没有参数。如果Student类从基类Person那里继承了writeOutput方法，Student就将包含两个都名为writeOutput的方法，并且都没有参数。Java有个规则避免这种问题。如果派生类定义了一个方法，其名字和基类中的一个方法相同，并且那个方法与基类中的方法有同样数量和类型的参数，则在派生类中的方法被称为**重写**（override）了基类中的方法定义。换句话说，派生类的对象使用的是派生类中的定义。

例如，图7-4中，下面的代码中对Student类的对象s进行的writeOutput方法调用将使用Student类中定义的writeOutput，而不是Person类中的定义：

```
s.writeOutput();
```

当重写方法时，可以对方法定义的主体进行任意的修改，但是不能修改方法的头部。尤其在重写方法时，不能修改方法的返回类型。

快速参考：重写方法定义

如果在派生类中有方法定义同基类中已有方法的名称相同，并且参数的类型和数量都相同，那么对派生类来说，新的方法定义替换了原有定义。

在此情况下，重写的方法定义必须同基类中方法的返回类型相同。即当重写方法定义时，不能改变方法的返回类型。

7.1.3 比较重写和重载

别把方法重写与方法重载相互混淆。当重写方法定义时，派生类中新的方法定义具有同样数目和类型的参数。另一方面，如果派生类中方法的参数数目或类型与基类中的方法不同，则派生类将同时有这两个方法，即重载。例如，假设在图7-3的Student类的定义中添加如下的方法：

```
public String getName(String title)
{
    return (title + getName());
}
```

在这种情况下，Student类就有两个名为getName的方法：它将从图7-1所示的Person类那里继承没有参数的getName方法，并且也拥有这个刚定义的、具有一个参数的getName方法。这是因为这两个都叫getName的方法有不同数目的参数，所以方法被重载了。

若弄不清重载和重写，有个小小的安慰，那就是它们都是合法的。

7.1.4 final修饰符

如果想指定某个方法定义不允许被派生类用新的定义重写，可以在方法的头部添加final修饰符，例如：

```
public final void specialMethod()
{
        .
        .
        .
```

用户一般不大可能会需要这个修饰符，不过在标准库中某些方法的规范说明里很可能会碰到它。

如果方法被声明为final，编译器对此方法被使用的方式就有了更多的了解，从而可以为该方法生成更高效的代码。

整个类可以被声明为final，这样它就不能被用作基类来派生任何类了。

▲ 易犯错误：使用基类中的私有实例变量

图7-3中的Student类的对象从图7-1的Person类中继承了名为name的实例变量。例如，下面的代码将把对象joe的实例变量name的值设置为"Josephine"（同时也把实例变量studentNumber设置为9891）：

```
Student joe = new Student("Josephine", 9891);
```

如果要改变joe.name（及joe.studentNumber），代码如下：

```
joe.reset("Joesy", 9892);
```

但是在处理像name这样的继承来的实例变量时，却需要仔细一些。Student类的实例变量name是从Person类继承而来的，但是实例变量name在Person类的定义中是私有实例变量，这就意味着name只能在Person类的方法定义中才可以直接访问。*不能在任何别的类的方法定义中通过名字来访问基类中的私有实例变量或方法，就算是派生类中的方法定义也不行。*

例如，下面的代码是Student类定义中reset方法的定义：

```
public void reset(String newName, int newStudentNumber)
```

```
    {
        setName(newName);
        studentNumber = newStudentNumber;
    }
```

合法的定义

你可能会奇怪为什么要用setName方法来设置实例变量name的值，如下编写此方法可行么：

```
    public void reset(String newName, int newStudentNumber)
    {
        name = newName; //ILLEGAL!
        studentNumber = newStudentNumber;
    }
```

非法的定义

正如其注释中指出的，这样行不通。实例变量name是Person类中的私有实例变量，尽管派生类如Student继承了name，但仍不能直接访问该变量，而是要通过某些公开的设置（或访问）方法。完成Student类的reset定义的正确途径是用setName方法来设置实例变量name。

基类的私有实例变量不能被派生类中的方法定义访问，这个事实经常让人无法接受。毕竟，学生应当能够改变他们自己的名字，而不应当被告知："抱歉，name是Person类的私有实例变量。"如果你是一个学生，你当然也是人。在Java中，这当然也对，Student类的对象也是Person类的对象。但是，私有实例变量和方法的使用规则必须是前面所描述的那样，否则就没有意义了。如果在派生类的方法定义中可以访问基类中的私有实例变量，那么在需要访问私有的实例变量的时候，只要简单地创建一个派生类，在该类的方法里访问那个变量就可以了。这就意味着只需要花一点功夫，任何人都可以访问所有的私有实例变量了。△

● 编程提示：假设你的同事图谋不轨

在前面的"易犯错误"部分中，提到私有的实例变量不能被派生类通过名字直接访问的原因是，如果不这样，恶意的程序员可以利用诡计来访问它们。你可能会争辩说自己的同事不可能想干坏事。实际上在初级课程中，有时候你是一项任务的唯一程序员，而自己是不可能试图故意搞砸自己的工作的。这些观点看起来不错，只是同事（有时候也包括自己）可能不经意间做了，虽说不是故意搞阴谋，但仍会带来问题。假设有心怀鬼胎的程序员，不是说同事们可疑，而是因为这是一种最好的方式来预防本是好心的程序员（也包括自己）犯错误。

▲ 易犯错误：私有方法未被继承

在前面的易犯错误小节中，我们强调了基类中的私有实例变量（或方法）不能被任何其他类的方法的定义所访问，*就算是在派生类的方法定义中也不可以*。注意，私有方法和私有的实例变量一样不能被直接访问。这常称为"私有方法未被继承。"这种说法未必绝对准确，不过确实抓住了精髓。私有的方法仍在那里，若某个公有的方法中有对私有方法的调用，该调用仍然能够奏效。但是，在派生类的方法定义中，不能包含对基类的私有方法的调用。

这不是什么问题。私有方法应当作为辅助方法使用，其应用应该限于定义它们的类本身。如果想让某个方法作为辅助方法为多个派生类服务，则它已经不仅仅是个辅助方法了，应该将其作为公有的方法。△

7.1.5 UML继承类图

图7-5显示了图7-2中的层次图的一部分，不过这次用的是UML表示法。注意，类图并不

完整。通常只需要显示手头的设计中需要的部分类图。图7-5和图7-2中表示法的唯一显著不同是图7-5中表示继承的线带有箭头。

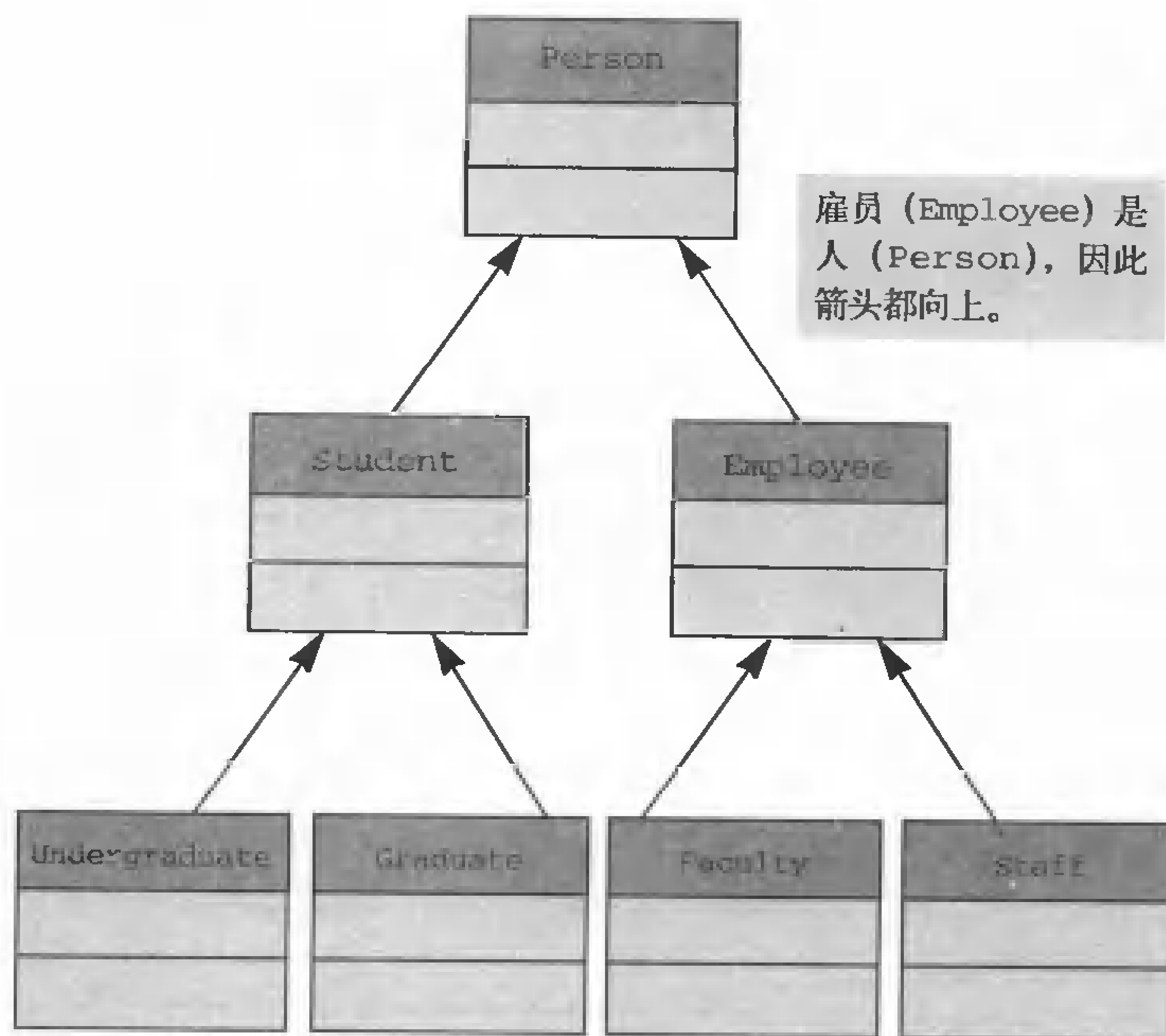


图7-5 UML表示法表示的类层次图

注意，箭头从派生类指向基类。这是因为箭头表示"是一个"关系。例如，一个Student是一个Person。用Java的术语，Student类型的对象也是Person类型的对象。

箭头也有助于定位方法定义。若查找某个类的方法定义，箭头指出了读者（或计算机）应当遵循的路径。如果要查找Undergraduate类的对象用的方法定义，首先在Undergraduate类的定义中查找；如果不在其中，就到Student类的定义中查找；如果仍然不在，就到Person类的定义中找。

图7-6显示了Person类及其派生类Student的继承层次的更多细节。假设s是Student类的对象。图7-6中的图可以指导读者找到下面的方法的定义：

```
s.writeOutput();
```

以及

```
s.reset("Josephine Student", 1234);
```

是在Student类中定义的，但

```
s.setName("Joe Student");
```

的定义是在Person的定义中。

自测题

1. 假设名为SportsCar的类是Automobile类的派生类。假设Automobile类有名为speed、manufacturer及numberOfCylinders的实例变量。SportsCar类的对象是否有名为speed、manufacturer及numberOfCylinders的实例变量？

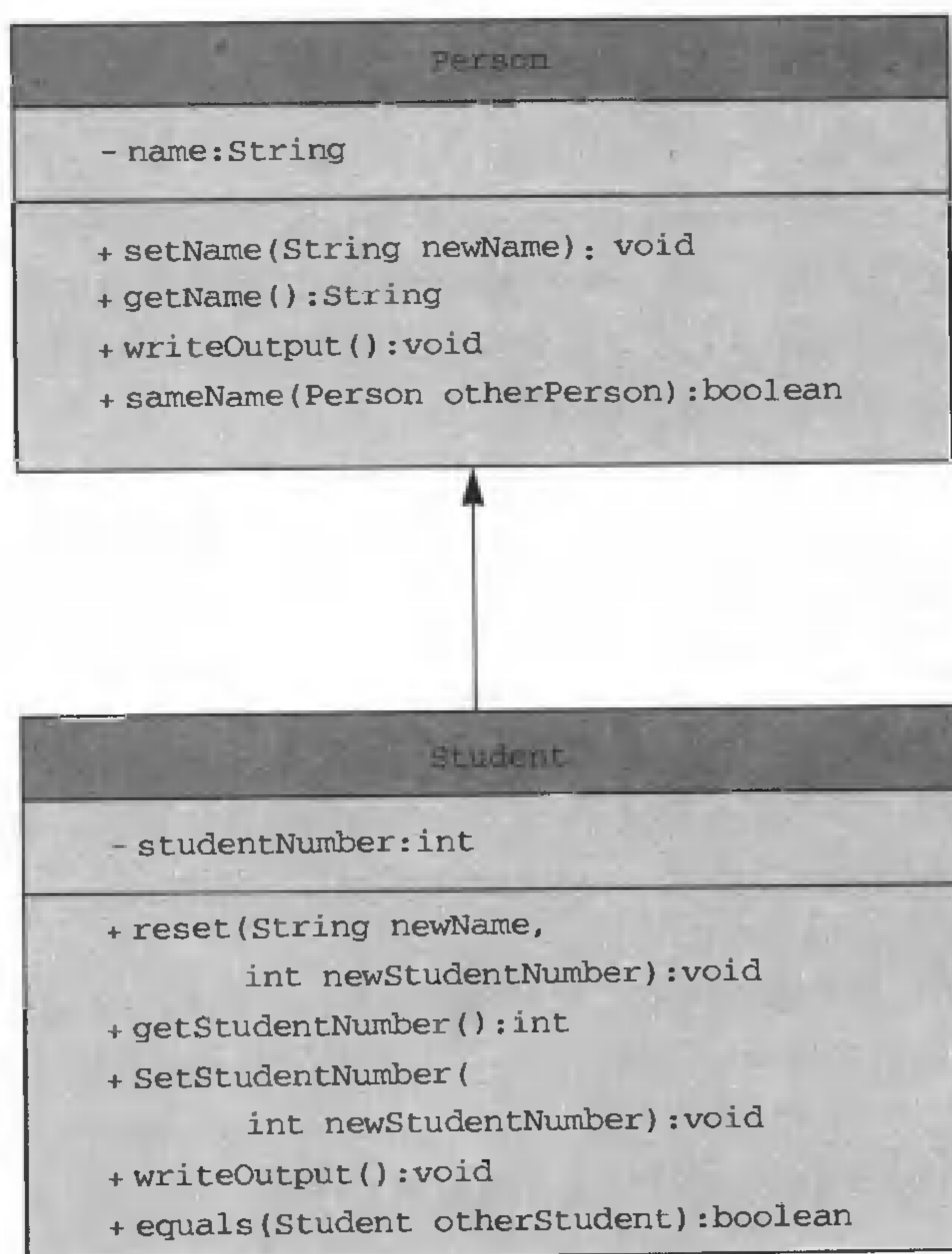


图7-6 UML类层次图的某些细节

2. 假设名为SportsCar的类是Automobile类的派生类，再假设Automobile类有公有方法，名为accelerate和addGas。SportsCar类的对象是否有accelerate和addGas方法？如果有，这些方法在SportsCar类和Automobile类中是否必须执行相同的动作？
3. 如果定义一个派生类，能否直接访问基类的私有实例变量？
4. 如果定义一个派生类，能否调用基类的私有方法？
5. 假设s是Student类的对象。根据图7-6中的继承类图，在哪里能找到下面代码中调用的sameName方法的定义？请解释答案。

```
Student other = new Student("Joe Student", 7777);
if (s.sameName(other))
    System.out.println("wow");
```

6. 假设s是Student类的对象。根据图7-6中的继承类图，在哪里能找到下面的代码中调用的方法的定义？并请解释答案。

```
s.setStudentNumber(1234);
```

7.2 使用继承编程

为了放弃继承权，你不必非死不可。

——某个不动产规划研讨会的广告

本节介绍定义和使用派生类时需要的一些基本技术。

7.2.1 派生类中的构造器

派生类（比如图7-3中的Student类）有自己的构造器。它所继承的基类（如Person）也有自己的构造器。在定义派生类的构造器时，通常第一步就是调用基类的一个构造器。例如，为Student类定义一个构造器。其中一个需要初始化的内容就是学生的名字。对名字的初始化通常在基类Person的构造器中完成（因为实例变量name是在Person的定义中引入的）。下面的代码是派生类Student的构造器，摘自图7-3：

```
public Student(String initialName, int initialStudentNumber)
{
    super(initialName);
    studentNumber = initialStudentNumber;
}
```

其中的代码行

```
super(initialName);
```

是对基类构造器的调用——在这里是调用Person类的构造器。注意，要用关键字super来调用基类的构造器，而不能使用其名称——也就是不能像下面这样：

```
Person(initialName); //ILLEGAL
```

关于super的使用有很多繁文缛节：它必须是构造器中的第一个动作，此后就不能使用。如果在构造器中没有调用基类的构造器，Java会自动添加对基类的默认构造器的调用作为派生类中的第一个动作。

下面的代码是Student类的构造器定义，摘自图7-3：

```
public Student()
{
    super();
    studentNumber = 0;//Indicating no number yet
}
```

该定义和下面的代码完全等价：

```
public Student()
{
    studentNumber = 0;//Indicating no number yet
}
```

快速参考：调用基类的构造器

定义派生类的构造器时，可以使用super作为基类构造器的名称。对super的调用必须是此构造器的第一个动作。

举例：

```
public Student(String initialName, int initialStudentNumber)
  {
      super(initialName);
      studentNumber = initialStudentNumber;
  }
```

7.2.2 this方法（选读）

定义构造器时，另一种常见的方式是调用同一个类的其他构造器。可以类似使用super

那样使用关键字this，但是用this，调用的是同一个类的构造器，而不是基类的构造器。例如，下面的代码定义了一个构造器，可以考虑将其加入图7-3的Student类中：

```
public Student(String initialName)
{
    this(initialName, 0);
}
```

这个构造器的定义中的唯一语句是对另一个构造器的调用，那个构造器的定义的开头如下：

```
public Student(String initialName, int initialStudentNumber)
```

同super一样，对this的使用必须放在构造器的第一个动作中。这样，构造器的定义中就不能既调用super又调用this了。如果想这么做怎么办？这种情况下，可以调用this，并在那个被调用的构造器中使用super作为其第一个动作。

快速参考：调用同一个类中的另一个构造器（选读）

定义某个类的构造器时，可以用this作为同一个类的另一个构造器的名称。对this的任何调用必须是该构造器的第一个动作。

举例：

```
public Student(String initialName)
{
    this(initialName, 0);
}
```

7.2.3 调用被重写的方法

定义派生类的构造器时，可以用super作为基类的构造器名称。还可以用super调用基类中被派生类重写（重定义）的方法，不过方式略有不同。

例如，图7-3中Student类的writeOutput方法。使用如下的代码输出Student的名字：

```
System.out.println("Name: " + getName());
```

可以用另一种方式，只要调用图7-1中Person类的writeOutput方法就可以输出名字，因为Person类的writeOutput方法会输出人的名字。唯一的问题是如果用了writeOutput这个Student类中已有的方法名作为名字，它将表示Student类中的writeOutput方法。需要的是一种说明“writeOutput()是在基类中定义的那个方法”的方式。用来表示这个意思的语法是super.writeOutput()。因此Student类的writeOutput方法的另外一种定义如下所示：

```
public void writeOutput()
{
    super.writeOutput();
    System.out.println("Student Number: " + studentNumber);
}
```

如果把图7-3中Student定义中的writeOutput定义替换成上面的代码，则Student的表现将和以前完全一样。

常见问题：如何调用被重写的方法的老版本？

在派生类的方法定义中，可以通过在方法名称前面加上super和点来调用基类中被重写的方法。

举例：

```
public void writeOutput()
{
    super.writeOutput();
    System.out.println("Student Number: " + studentNumber);
}
```

编程示例：多级派生类

可以用派生类构造派生类。实际上这很常见。例如，在图7-7中定义了Undergraduate类，它是图7-3中的Student 类的派生类。这意味着Undergraduate类的对象有Student类的所有方法和实例变量。但是Student已经是图7-1中Person的派生类了，所以这也意味着Undergraduate的对象有Person类的所有方法和实例变量。Person类的对象有实例变量name。Student类的对象有实例变量name和studentNumber。Undergraduate类的对象有实例变量name、studentNumber以及添加的实例变量level。Undergraduate类的对象必须用访问和设置方法来访问实例变量name和studentNumber，不过它拥有这些实例变量这是毫无疑问的。

```
public class Undergraduate extends Student
{
    private int level; //1 for freshman, 2 for sophomore, //3 for junior, or 4 for
    senior.
    public Undergraduate()
    {
        super();
        level = 1;
    }
    public Undergraduate(String initialName, int initialStudentNumber, int
    initialLevel)
    {
        super(initialName, initialStudentNumber);
        setLevel(initialLevel); //Checks 1 <= initialLevel <= 4
    }

    public void reset(String newName, int newStudentNumber, int newLevel)
    {
        reset(newName, newStudentNumber);
        setLevel(newLevel); //Checks 1 <= newLevel <= 4
    }

    public int getLevel()
    {
        return level;
    }

    public void setLevel(int newLevel)
    {
        if ((1 <= newLevel) && (newLevel <= 4))
            level = newLevel;
```

图7-7 派生类的派生类

```
        else
        {
            System.out.println("Illegal level!");
            System.exit(0);
        }
    }

    public void writeOutput()
    {
        super.writeOutput();
        System.out.println("Student Level: " + level);
    }

    public boolean equals(Undergraduate otherUndergraduate)
    {
        return (super.equals(otherUndergraduate)
            && (this.level == otherUndergraduate.level));
    }
}
```

图7-7 （续）

图7-8中的UML类图显示了Person、Student及Undergraduate这些类之间的关系。

这样的派生类的链可以很有效地对代码进行复用。Undergraduate和Student（以及从它们派生出的其他类）实际上复用了Person类的定义中的代码，因为它们继承了Person类的全部方法。

Undergraduate类的构造器都以对super的调用开始，在此它表示基类Student的构造器。但是Student类的构造器同样以对super的调用开始，它表示基类Person的构造器。因此，当使用new调用Undergraduate类的构造器时，首先调用Person类的构造器，其次调用Student类的构造器，接着调用Undergraduate类的构造器中super之后的代码。

Undergraduate类的reset方法的定义如下：

```
public void reset(String newName,
                        int newStudentNumber, int newLevel)
{
    reset(newName, newStudentNumber);
    setLevel(newLevel); //Checks 1 <= newLevel <= 4
}
```

注意，这个方法以对只有两个实参的reset的调用开始。这会调用基类Student的名为reset的方法。在Undergraduate类中，名为reset的方法是重载的，因为这两个reset方法的实参列表是不同的，一个有两个实参，而另一个有3个实参。有两个实参的方法是从Student类中继承来的，不过它仍然是Undergraduate类的货真价实的方法。

3个实参的reset（在Undergraduate类中定义，参见前面显示的代码）方法重置实例变量name和studentNumber的值，分别设为newName和newStudentNumber，它使用下面的代码：

```
reset(newName, newStudentNumber);
```

通过调用setLevel，新的实例变量level被重置为newLevel。

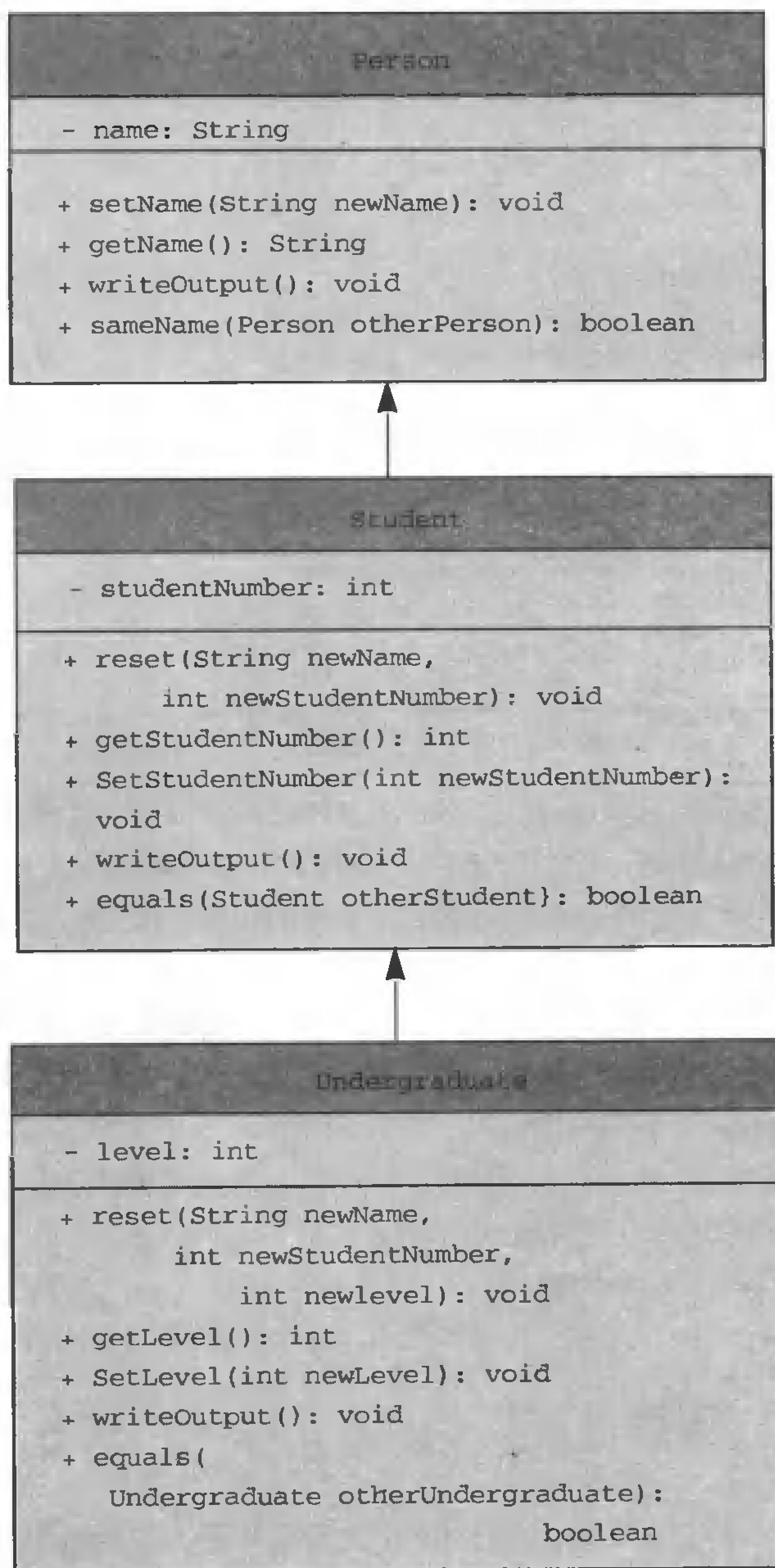


图7-8 UML类层次图中的某些更多细节

记住在Undergraduate类的定义中，不能通过名字来访问name和studentNumber（分别是基类Person和Student的私有实例变量），因此需要用设置方法来修改这些变量。Student类的reset方法就是上佳之选。

与Undergraduate类的reset方法的定义相对比，注意下面列出的writeOutput方法的定义：

```
public void writeOutput()
{
```

```
        super.writeOutput();
        System.out.println("Student Level: " + level);
    }
```

reset的定义是重载的示例。而writeOutput方法是重写的示例。

Undergraduate类中定义的reset和Student类中定义的reset的形参数目不同。 因此在派生类Undergraduate中同时有这两个版本的reset不会发生冲突。这是重载的示例。作为对比，Undergraduate类中定义的writeOutput方法与基类Student中的同名方法有相同的形参列表。这样，当调用writeOutput时，Java必须判断要使用哪一个writeOutput的定义。对派生类Undergraduate的对象来说，将使用Undergraduate类定义中的writeOutput版本。这就是在派生类中对名为writeOutput的方法进行重写的示例。基类Student中给出的writeOutput方法定义仍被派生类Undergraduate所继承。但是要在派生类Undergraduate的定义中调用基类Student定义的writeOutput版本，就要在方法名writeOutput前面加上super和一个点，就像前面显示的派生类Undergraduate定义中的writeOutput方法定义的第一行。

在图7-7中可以看到，equals方法定义中对super的类似用法。注意，这样做绝对合法，并且在派生类中对名为writeOutput和equals这样的方法进行重写是很普遍的。基类的版本仍然在那里，可以通过加上super前缀和一个点来访问它。

关于如何使用Undergraduate类，没什么特别的，不过还是在本书网站上的源代码文件UndergraduateDemo.java中包含了一个简单的演示程序。

快速参考：父类和子类

在讨论派生类时，经常使用从家庭关系演化来的术语。基类常被称为**父类**，派生类则被称为**子类**。这样称呼使描述继承关系的语言非常流畅。例如，说子类从父类继承了实例变量和方法。这种比拟常会进一步深入下去。另一个类的父类的父类的父类（或其他数量的父辈迭代关系）常被称为**祖先类**。若A类是B类的祖先，则B类通常称为A类的子孙。

7.2.4 关于重载和重写的一个微妙之处（选读）

考察Student类和Undergraduate类中的equals方法。它们的参数列表不同。Student类中的equals方法有一个Student类型的形参，而Undergraduate类中的equals方法则有一个Undergraduate类型的形参。它们有同样数量（1个）的形参，不过这个形参在两者的定义中是不同的类型。类型上的不一致足以符合重载的条件了（前面已经提到，两个方法定义需要有不同数目的形参或者某个位置上的形参类型不同，就符合重载的条件）。因此，从技术角度来说，派生类Undergraduate中equals的第二个定义是重载而不是重写。

那为什么在派生类Undergraduate的equals定义中要用super呢？这个定义（用在派生类Undergraduate中）如下：

```
public boolean equals(Undergraduate otherUndergraduate)
{
    return (super.equals(otherUndergraduate)
        && (this.level == otherUndergraduate.level));
}
```

既然super.equals调用的是重载的equals，为什么还需要用super呢？省略它，通过下面的代码就可以看出为什么了：

```
return (equals(otherUndergraduate)
          && (this.level == otherUndergraduate.level));
```

实参otherUndergraduate的类型是Undergraduate，因此Java将假设这里要用的是Undergraduate类中的equals定义。为了强制Java使用基类Student中的equals定义，就要用super和点。

■ Java提示：不能使用多重super

在派生类的方法定义中，在方法名前面加上super和点，就可以调用基类中被重写的方法。不过，除了直接父辈，其他祖先类的方法不能通过反复地使用super来调用。假设Student类是从Person类派生所得，Undergraduate类从Student类派生所得。你可能想用super.super在Undergraduate类的方法定义中调用Person类的方法，例如：

```
super.super.writeOutput();//ILLEGAL!
```

不过，正如注释中指出的，在Java中这样的super序列是不合法的。

● 编程提示：对象可以有多个类型

图7-7中的Undergraduate类是Student类的派生类。在现实世界里，每个本科生都是学生。这种关系在Java中也正确。每个Undergraduate类的对象同时也是Student类的对象。这样，如果有个方法，其形参的类型是Student，则调用这个方法时所用的实参就可以是Undergraduate类的对象。在这个例子中，该方法只可以使用属于Student类的实例变量和方法，但是每个Undergraduate类的对象都具有这些实例变量和方法，因此该方法还是可以对Undergraduate类的对象进行很多有意义的操作。

例如，假设Student类和Undergraduate类在图7-3和图7-7中定义，下面是可能在某个类中出现的方法定义：

```
public class SomeClass
{
    public static void compareNumbers(Student s1, Student s2)
    {
        if (s1.getStudentNumber() == s2.getStudentNumber())
            System.out.println(s1.getName()
                        + " has the same number as "
                        + s2.getName());
        else
            System.out.println(s1.getName()
                        + " has a different number than "
                        + s2.getName());
    }
  ...
}
```

使用这个方法的程序可能包含如下的代码：

```
Student studentObject = new Student("Jane Doe", 1234);
System.out.println("Enter name:");
Scanner keyboard = new Scanner(System.in);
String undergradName = keyboard.nextLine();
System.out.println("Enter student number :");
```

```
    int undergradStudentNumber = keyboard.nextInt();
    Undergraduate undergradObject =
        new Undergraduate(undergradName, undergradStudentNumber, 1);
    SomeClass.compareNumbers(studentObject, undergradObject);
```

在compareNumbers的方法首部，可以看到两个参数都是Student类型。但是，如下的调用：

```
SomeClass.compareNumbers(studentObject, undergradObject);
```

用Student类型的对象作为一个实参，而另一个实参是Undergraduate类型的对象。在实参的类型是Student时，怎么能用Undergraduate类型的对象呢？答案就在于每个Undergraduate类型的对象也是Student类型的。为了表现得更清晰，请看下面的代码，它交换了这两个实参，而这个方法调用仍是合法的：

```
SomeClass.compareNumbers(undergradObject, studentObject);
```

注意这里没有自动类型转换。Undergraduate类的对象是Student类的一员，因此也是Student类型。它不需要，也没有被转换为Student类的对象。

对象可以实际拥有多个类型是继承的结果。回想图7-7中的Undergraduate类，它是图7-3中的Student类的派生类，而Student是图7-1中的Person类的派生类。这意味着每个Undergraduate类的对象也是Student类型的对象及Person类型的对象。这样，对Person类的对象适合的操作，对Undergraduate类的对象也都适合。

例如，假设Person类和Undergraduate类定义在图7-1和图7-7中，下列是某个程序中可能出现的代码：

```
Person joePerson = new Person("Josephine Student");
System.out.println("Enter name:");
Scanner keyboard = new Scanner(System.in);
String newName = keyboard.nextLine();
Undergraduate someUndergrad = new Undergraduate(newName, 222, 3);
if (joePerson.sameName(someUndergrad))
    System.out.println("Wow, same names!");
else
    System.out.println("Different names");
```

从图7-1中sameName方法的头部可以看出，它有一个Person类型的参数。然而，尽管实参someUndergrad是Undergraduate类的对象，也就是说它的类型是Undergraduate，而且该参数适用于Person类型，前面的if-else语句中用到的下面的调用是绝对合法的：

```
joePerson.sameName(someUndergrad)
```

这是因为Undergraduate类的每个对象都是Person类的对象。甚至下面的代码也是合法的：

```
someUndergrad.sameName(joePerson)
```

sameName方法需要Person类型的调用对象，而someUndergrad是Undergraduate类型，但是没问题。Undergraduate类型的对象也具有Person类型。对祖先类的对象有效的操作也适用于子孙类。

你可能已经明白，如果A是B类的派生类，而B是C类的派生类，那么A类的对象就是A类型的。它也是B类型，同时也是C类型。这对任意的派生类链都成立，不管链有多长。

由于派生类的对象具有所有其祖先类的类型（也包括其自身的类型），可以把某类的对象赋值给任何祖先类型的变量，但是反之不行。例如，若Student是Person的派生类，而Undergraduate是Student的派生类，则下列代码是合法的：

```
Person p1, p2;
Student s = new Student();
Undergraduate ug = new Undergraduate();
p1 = s;
```

```
    p2 = ug;
```

甚至可以不用变量s和ug，直接把新对象赋给变量p1和p2，代码如下：

```
    Person p1, p2;
    p1 = new Student();
    p2 = new Undergraduate();
```

不过，下面的代码就不合法了：

```
    Student s = new Person(); //ILLEGAL!
    Undergraduate ug = new Person(); //ILLEGAL!
    Undergraduate ug2 = new Student(); //ILLEGAL!
```

下面这些代码，看起来似乎没问题，可同样是非法的：

```
    Undergraduate ug, ug2; //valid
    Person p = new Person(); //valid
    ug = p; //ILLEGAL!
    Student s = new Student(); //valid
    ug2 = s; //ILLEGAL!
```

这些都非常合理。例如，学生（Student）是人（Person），但是人（Person）不一定是学生（Student）。有些程序员发现“是一个”这种关系短语在判断对象可以是什么类型的，以及赋值给变量是否合法方面很有用。

另一个例子，若Employee是Person的派生类，那么雇员（Employee）是一个人（Person），因此可以将Employee对象赋给Person类型的变量。反过来，一个人（Person）不一定是一个雇员（Employee），因此不能把从Person类创建的对象赋值给Employee类型的对象。

记住：派生类的对象有多个类型

派生类的对象具有派生类的类型，并且也具有基类的类型。更一般地说，派生类的对象具有所有祖先类的类型。

快速参考：赋值兼容性

可以把派生类的对象赋值给任何祖先类的变量，但是反之不行。

● 编程提示：“是一个”（is a）与“有一个”（has a）关系

前面已经说明了，一个学生（Student）是一个人（Person），因此把Student类作为Person类的派生类。这是类之间的“是一个”关系的例子。这是从简单的类构造复杂类的一种途径。

还有一种办法可以从一个简单的类构造更复杂的类，是利用“有一个”关系。例如，若有个Date类可以记录日期，就可以向Student类中添加一个Date类型的实例变量来记录其注册日期。在这种情形下，说一个Student“有一个”Date。再看一个例子，若有一个机械臂（MechanicalArm）类，而我们要定义一个模拟机器人的类，可以让机器人（Robot）类有一个MechanicalArm类型的实例变量。这时说一个Robot“有一个”MechanicalArm。

大多数情况下，代码用“是一个”关系或“有一个”关系来组织，都可以工作。让Robot类是MechanicalArm类的派生类看起来很蠢（也的确很傻），但是确实可以这么做，而且可以设法使代码运行（不过可能比较困难）。幸运的是，最佳的编程技巧通常是简单遵循在日常用语中看起来最自然的方式。说“一个机器人有一个机械臂”比说“一个机器人是一个机械臂”要更地道一些，因此让Robot类有一个MechanicalArm类型的实例变量在编程中更合理。

在编程技术的文献中，术语“是一个”和“有一个”很常见。

自测题

7. 给出TitledPerson类的完整定义。TitledPerson是图7-1中的Person类的派生类。TitledPerson类还有个额外的String类型的实例变量，用来记录头衔，如“女士”（Ms）、先生（Mr.）或阁下（The Honorable）。TitledPerson类有两个构造器，即默认的构造器及另一个设置姓名和头衔的构造器。它有writeOutput方法、reset方法、equals方法，返回头衔的访问方法getTitle及修改个人头衔的设置方法setTitle。两个有头衔的人是否相等，取决于其姓名及头衔。可以使用图7-3中的Student类作为范本。
8. 重写图7-7中的Undergraduate类的writeOutput方法的定义，使用getName和getStudentNumber而不是super.writeOutput。（大多数程序员会用图7-7中的版本，但是应当能够写出这两种不同的版本。）
9. 重写图7-7中的Undergraduate类的reset方法的定义，使用setName和setStudentNumber而不是被重载的reset方法。（大多数程序员会用图7-7中的版本，但是读者应当能够写出这两种不同的版本。）
10. 一个对象能够有多个类型吗？
11. 下面的语句创建的对象是哪种或哪些类型的？（Undergraduate类的定义在图7-7中给出。）
```
Undergraduate ug = new Undergraduate();
```
12. 描述关键字super的两种用途。
13. （如果阅读过7.2.2节，回答本题。）在构造器的定义中，this和super都被用于表示所调用的方法的名称，它们有何不同？

7.2.5 Object类

Java有一个“夏娃”（Eve）类——也就是说，该类是所有其他类的祖先。在Java中，每个其他类都是Object类的派生类（或是其派生类的派生类等）。因此，每个类的每个对象在具有其本身的类型的同时，都具有Object类型（以及该类的所有祖先类的类型）。甚至你自己定义的类也是Object类的后继类。如果没有把类作为其他类的派生类，Java就会自动把它作为Object类的派生类。

快速参考：Object类

在Java中，每个类都是预定义的Object类的子孙类。因此每个类的每个对象在具有其本身的类型的同时，都具有Object类型（以及该类的所有祖先类的类型）。

有了Object类，程序员就能够写出Java方法代码，其参数（使用Object类型）可以被任何类的对象取代。你会碰到一些库方法，这些方法接受Object类型的实参，因此可以用任何类的对象作为实际的实参。

Object类还有一些方法能供所有的Java类继承。例如，每个对象都直接或间接地从Object类继承了方法equals和toString。然而，所继承的equals和toString几乎对你自己定义的任何类都不太合适。必须用新的更合适的定义来重写这些方法。

（写出正确的equals方法对初入此道的程序员来说要求过高了。本书中提供的equals方法在大多数情况下工作良好。若读者想了解如何写出完整和正确的equals，请看7.3节的Java提示（选读）“更好的equals方法”。）

继承来的toString方法没有实参。toString方法应当把对象中的数据都编码进一个String中作为返回值。但是，继承来的toString方法几乎没有什么用处，因为它不可能产生一个能很好地表达数据的字符串。需要重写toString的定义使其为所定义的类的对象中的数据产生合适的String来。

例如，下列代码中的toString定义可以被加入图7-3中的Student类：

```
public String toString()
{
    return("Name: " + getName()
                    + "\nStudent number: "
                    + studentNumber);
}
```

如果Student类中加入这个toString方法，它就可以和别的对象一样，用下面的代码产生输出：

```
Student s = new Student("Joe Student", 2001);
System.out.println(s.toString());
```

输出如下：

```
Name: Joe Student
Student number: 2001
```

toString方法有一个很特别的性质。如果没有在对System.out.println的调用中使用这个方法，它会被自动调用，即下面的代码是等价的：

```
System.out.println(s.toString());
System.out.println(s);
```

这两个语句将产生如下输出：

```
Name: Joe Student
Student number: 2001
Name: Joe Student
Student number: 2001
```

因此，为自己的类写出合适的toString方法会是个不错的主意。

关于toString和System.out.println的更多交互细节，参见7.3.3节。

可以从Web上得到的本书源代码中给出的Student类里有这个toString方法，因此可以很容易地尝试一下。本书姊妹篇《Java程序设计与问题解决：高级篇（第4版）》第1章的Species类也有toString方法，若想试试这个版本，打开Species.java文件，它存放于《Java程序设计与问题解决：高级篇（第4版）》第1章的源代码中。此时可以忽略其中的"implements Serializable"或者把这个子句注释掉，在《Java程序设计与问题解决：高级篇（第4版）》第1章中会对这个子句进行解释。

另一个从Object类中继承的方法是clone。该方法无实参，返回调用对象的一个副本。返回的对象应当和调用的对象有一样的数据内容，但却是不同的对象（双胞胎的另一个或一个"克隆"）。与其他从Object类中继承的方法一样，clone方法也需要重新定义（重写）才能正常运作。不过，对于clone方法，还有别的事物需要一并处理。在第5章的易犯错误节"隐私泄露"和附录H中对该方法进行了一些讨论，然而对clone的彻底研究已超出了本书的范围。

自测题

14. 考察已在前一节中讨论过的如下代码：

```
Student s = new Student("Joe Student", 2001);
System.out.println(s.toString());
```

为何输出在两行中而不是都在一行里？

15. 下面的代码行哪些合法而哪些不合法？（Student是Person的派生类，Undergraduate是Student的派生类。）

```
Person p1 = new Student();
Person p2 = new Undergraduate();
Student s1 = new Person();
Student s2 = new Undergraduate();
Undergraduate ug1 = new Person();
Undergraduate ug2 = new Student();
Object ob = new Student();
Student s3 = new Object();
```

案例分析：字符图形

Java有些方法可以在终端的屏幕上绘图。但是，有时屏幕或者其他类型的输出设备上没有可绘图的能力，比如，某些老式的终端只能允许文本输出。

在本例中，将设计3个简单的类，仅仅使用文本在屏幕上产生简单的图形。这些类通过将普通的键盘字符放置在每行的特定位置上来绘出简单的图形。

下面详述本例中将要做的事情：创建两个类，一个画盒子，另一个画三角形。再写一个简单的演示程序使用三角形和盒子类来画出松树。

每个图形，如盒子和三角形，都有一个偏移值说明它离屏幕边缘缩进了多远。每个图形还有一个尺寸值，不过该尺寸对于盒子和三角形来说定义方法不同。

对盒子来说，尺寸用宽度和长度给出，*单位是字符的数目*。因为字符本身的纵向长度大于横向宽度，盒子在屏幕上比期望的要瘦高一些。例如，一个5×5的盒子在屏幕上看起来不是方的，而是如图7-9所示。

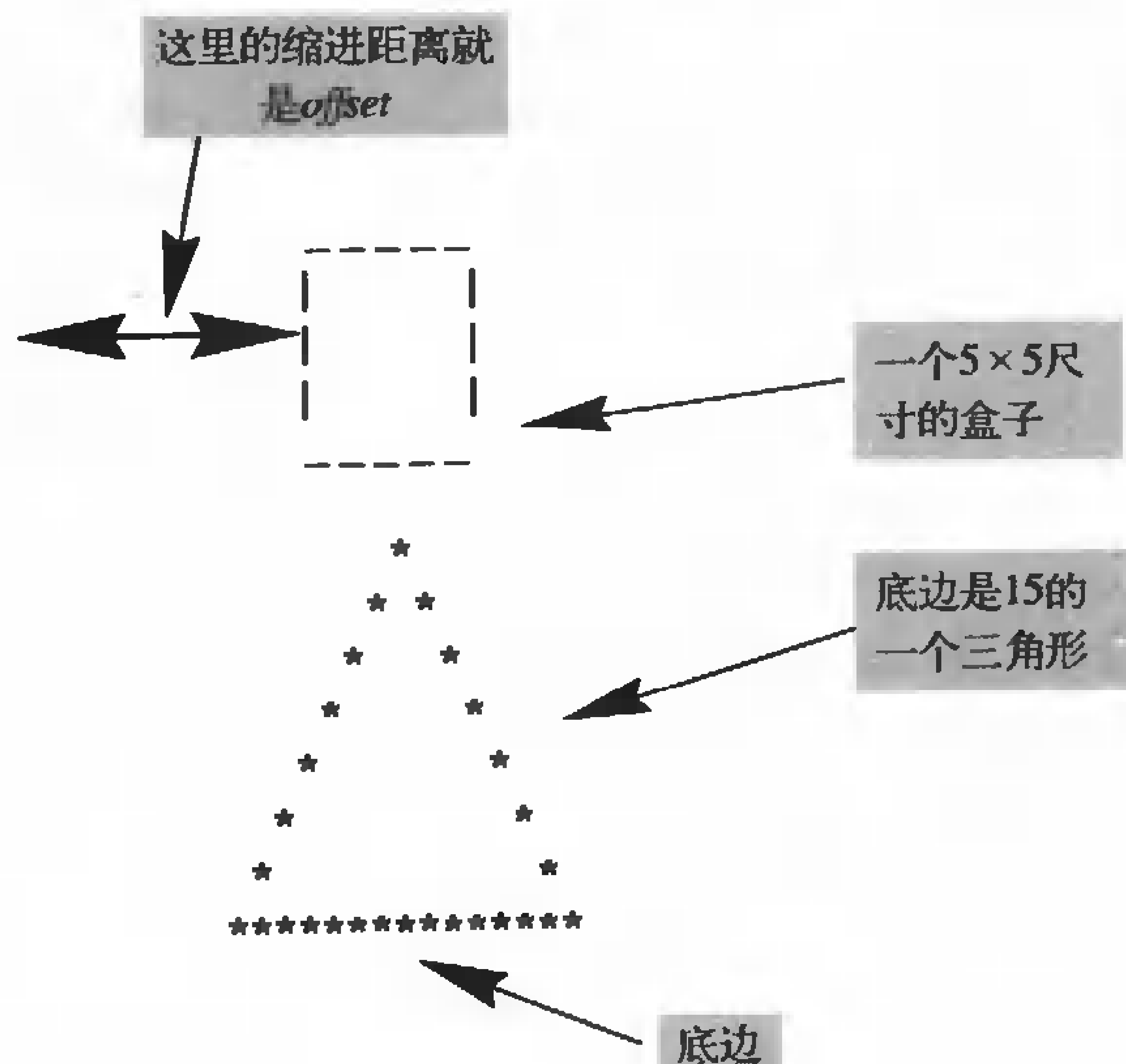


图7-9 盒子和三角形示例

对三角形来说，尺寸用底边的长来表示。三角形指向上方，底边位于下方。三角形的斜边受到每行一个字符的限制（只有这样才比较平滑），因此只要底边确定了，三角形的侧边就确定了。图7-9给出了盒子和三角形的示例。

因为盒子和三角形一般来说是具有很多相同属性的图形，应当设计一个基类名为Figure。那么Box类和Triangle类就是Figure类的派生类。Figure类将为所有的图形都普遍具备的属性设置实例变量，并为所有图形都需要的动作设置方法。这样，就有以下属性和动作。

属性（properties）。所有的图形都有一个偏移量，表示图形缩进的空白字符数目，因此这个偏移量可以存放在一个int类型的实例变量中。所有的图形都会有大小，但是有些图形的大小可以用单个数字来描述，而其他图形需要用两个或更多的数字才能定义大小。因此Figure类无法设置大小属性。Figure类将只有下面这个实例变量：

```
private int offset;
```

动作（actions）。所有的图形都需要的动作就是设置参数及画出图形。设置参数可以由构造器和setOffset方法处理。如此，定义的Figure类如图7-10所示。

drawHere方法只是简单地根据偏移量的值在屏幕上缩进一些空白，然后写一个星号。这只是为了可以有些内容供测试用。在实际的应用程序中是不会使用这种drawHere的。在定义盒子和三角形的类时就会重写drawHere的定义。

drawAt方法有一个int类型的参数。drawAt方法根据这个参数的值插入一些空行，然后调用drawHere画出图形。当然，在这里它表现平平，不过，重写drawHere之后，drawAt就能画出有意思的图形了。

下面集中精力学习画盒子的类。这个类名为Box，将是Figure类的派生类。你需要决定在Figure类的基础上增加哪些实例变量和方法，还要考虑Figure中的哪些方法定义需要通过重写来改变。

属性。Box类继承了offset实例变量，但还需要为盒子的高度和宽度添加实例变量。该类的定义如下：

```
public class Box extends Figure
{
    private int height;
    private int width;
    <仍然需要方法定义。>
}
```

注意这里没有列出实例变量offset，它是Box从基类Figure继承来的。

动作。Box类有普通的构造器和reset方法。它从Figure类继承了drawAt及drawHere方法。不过，需要重写drawHere方法以便它实际上画出一个盒子来。因此drawHere方法要放在需要定义的列表中。

下一步，考察一下drawAt方法。需要重写它吗？看图7-10中的drawAt方法，只要drawHere定义正确，drawAt方法对盒子或任何图形都可以很好地工作。所以不需要重新定义drawAt方法。只需要重定义drawHere方法。下面就应当将动作作为类的方法进行编码了。

首先看看示例的构造器。需要设计一个构造器把所有的实例变量设置为实参的值。但是那个名为offset的实例变量是基类Figure的私有实例变量，因此不能直接访问，不过可以调用基类的构造器super。这样这个构造器的代码如下：

```
public Box(int theOffset, int theHeight, int theWidth)
{
    super(theOffset);
    height = theHeight;
    width = theWidth;
}
```

```
/**
Class for simple character graphics figures to be sent to the screen. This class
will draw an asterisk on the screen as a test. It is not intended to be used as a
"real" figure in any graphics. It is intended to be used as a base class for other
classes of figures that will be used in graphics applications.
*/
public class Figure
{
    private int offset;

    public Figure()
    {
        offset = 0;
    }

    public Figure(int theOffset)
    {
        offset = theOffset;
    }

    public void setOffset(int newOffset)
    {
        offset = newOffset;
    }

    public int getOffset()
    {
        return offset;
    }

    /**
     Draws the figure at lineNumber lines down from the current line.
    */
    public void drawAt(int lineNumber)
    {
        int count;
        for (count = 0; count < lineNumber; count++)
            System.out.println();
        drawHere();
    }

    /**
    Draws the figure at the current line.
    */
    public void drawHere()
    {
        int count;
        for (count = 0; count < offset; count++)
            System.out.print(' ');
        System.out.println('*');
    }
}
```

图7-10 基类Figure

默认构造器及reset方法参见图7-11。注意，reset方法需要用专用的设置方法来重置从基类Figure继承来的私有实例变量offset。

```
/**
Class for a rectangular box to be drawn on the screen. Because each character is
higher than it is wide, these boxes will look higher than you might expect.
Inherits getOffset, setOffset, and drawAt from the class Figure.
*/
public class Box extends Figure
{
    private int height;
    private int width;

    public Box()
    {
        super();
        height = 0;
        width = 0;
    }

    public Box(int theOffset, int theHeight, int theWidth)
    {
        super(theOffset);
        height = theHeight;
        width = theWidth;
    }

    public void reset(int newOffset, int newHeight, int newWidth)
    {
        setOffset(newOffset);
        height = newHeight;
        width = newWidth;
    }

    /**
    Draws the figure at the current line.
    */
    public void drawHere()
    {
        drawHorizontalLine();
        drawSides();
        drawHorizontalLine();
    }
    private void drawHorizontalLine()
    {
        spaces(getOffset());
        int count;
        for (count = 0; count < width; count++)
            System.out.print('-');
        System.out.println();
    }

    private void drawSides()
```

图7-11 Box类

```
        {
            int count;
            for (count = 0; count < (height - 2); count++)
                drawOneLineOfSides();
        }

        private void drawOneLineOfSides()
        {
            spaces(getOffset());
            System.out.print('|');
            spaces(width - 2);
            System.out.println('|');
        }

        //Writes the indicated number of spaces.
        private static void spaces(int number)
        {
            int count;
            for (count = 0; count < number; count++)
                System.out.print(' ');
        }
    }
```

spaces方法被设为静态（static）的，因为它不需要一个调用对象。如果spaces不是static，这个类仍然可以运作良好，不过把它设为static会更加清晰一些。

图7-11 （续）

在Box类的默认构造器中，可以省略对基类构造器super()的调用，它会被自动调用的。不过为了清晰起见还是保留了这个调用。

这些构造器和专有访问方法的定义大部分都与任何类的定义类似。不过，drawHere方法的定义很大程度上依赖于图形该怎么画。可以使用著名的**自顶向下设计**（top-down design）方法来定义它。自顶向下设计的基本技术是把任务分解为子任务来执行。例如：

画盒子
(1) 画顶线
(2) 画边线
(3) 画底线

注意，不是所有选择子任务的方式都可以奏效。起先可能试图分解为两个子任务，盒子的左右两边各一个。然而，输出必须一行接一行地进行，不能回头，所以必须同时画两边（否则形状就会扭曲）。任务分解之后，定义出drawHere方法就很容易了：

```
public void drawHere()
{
    drawHorizontalLine();
    drawSides();
    drawHorizontalLine();
}
```

尽管很简单，但它确实把大部分工作都放到了后头。还需要定义drawHorizontalLine和drawSides方法。由于它们都是起辅助作用的，所以将被作为私有方法。

drawHorizontalLine的伪代码如下：

```
输出offset个空白字符。
输出width个'-'字符。
System.out.println();
```

drawHorizontalLine方法的最终代码如图7-11所示。注意写一定数目的空白的任务被独立出来作为另一个辅助方法，名为spaces。

接下来，我们把注意力集中到drawSides方法上。该任务需要画如下的图形：

```
|        |
|        |
|        |
|        |
```

注意，每行都是相同的，可以把写其中一行的工作提取为子任务。drawSides方法的定义如下：

```
private void drawSides()
{
    int count;
    for (count = 0; count < (height - 2); count++)
        drawOneLineOfSides();
}
```

注意，这里画的行数比高度少两行，顶和底两个水平行占用了两个高度单位。

这样就只剩下辅助方法drawOneLineOfSides需要定义了。它的伪代码如下：

```
spaces(getOffset());
System.out.print('|');
spaces(width - 2);
System.out.println('|');
```

因为已经有了编写空白的子任务的方法，上述伪代码就直接成为Java代码了，这样drawOne-LineOfSides就定义完毕。Box类的完整定义如图7-11所示。

尽管在本例中，不会讲述测试过程，但Figure、Box及Triangle（这个目前还没有讨论到）类的所有方法都是需要测试的。记住，每个方法应当在一个程序中进行测试，而在该程序中，只有这个方法是没有测试过的。

图7-12中给出了Triangle类的定义。可以用设计Box类相同的技术来设计该类。这里只讨论drawHere方法中的一个部分，你第一次阅读时可能对它的技术细节不太明白。drawHere方法把任务分为两个子任务：画倒置的V作为三角形的顶，以及画三角形底边的水平行。我们只讨论画倒置V的drawTop方法。

drawTop方法画出的图形如下：

```
      *
     * *
    *   *
   *     *
  *       *
 *         *
*           *
```

注意，整幅图有一个偏移量。图形最底边的缩进恰好等于这个偏移量，但是每向上一行，就增加一个缩进量。反之，从上向下（计算机必须这样），每行减少一个缩进量。每行的缩进减少一个字符，若用int类型的变量startOfLine的值来表示缩进量，缩进就可以用下面的代码实现：

```
spaces(startOfLine);
```

```
/**
Class for triangles to be drawn on the screen. For this class, a triangle points
up and is completely determined by the size of its base. (Screen character spacing
determines the length of the sides, given the base.)
Inherits getOffset, setOffset, and drawAt from the class Figure.
*/
public class Triangle extends Figure
{
    private int base;
    public Triangle()
    {
        super();
        base = 0;
    }

    public Triangle(int theOffset, int theBase)
    {
        super(theOffset);
        base = theBase;
    }

    public void reset(int newOffset, int newBase)
    {
        setOffset(newOffset);
        base = newBase;
    }

    /**
     Draws the figure at current line.
    */
     public void drawHere()
     {
         drawTop();
         drawBase();
     }

    private void drawBase()
    {
        spaces(getOffset());
        int count;
        for (count = 0; count < base; count++)
            System.out.print('*');
        System.out.println();
    }

    private void drawTop()
    {
        //startOfLine will be the number of spaces to the
        //first '*' on a line. Initially set to the number
```

图7-12 Triangle类

```
        //of spaces before the top '*'.
        int startOfLine = getOffset() + (base/2);
        spaces(startOfLine);
        System.out.println('*'); //top '*'

        int count;
        int lineCount = (base/2) - 1; //height above base
        //insideWidth == number of spaces between the
        //two '*'s on a line.
        int insideWidth = 1;
        for (count = 0; count < lineCount; count++)
        {
            //Down one line, so the first '*' is one more
            //space to the left.
            startOfLine--;
            spaces(startOfLine);
            System.out.print('*');
            spaces(insideWidth);
            System.out.println('*');
            //Down one line, so the inside is 2 spaces wider.
            insideWidth = insideWidth + 2;
        }
    }

    private static void spaces(int number)
    {
        int count;
        for (count = 0; count < number; count++)
            System.out.print(' ');
    }
}
```

图7-12 （续）

这可以用一个对行的循环来做，每次循环迭代中把startOfLine的值减少1。同一行中的两个星号的间隔每向下一行就增加2。若间隔用int类型的变量insideWidth的值来表示，则除了最顶行的星号之外，倒置V的绘图循环如下：

```
for (count = 0; count < lineCount; count++)
{
    spaces(startOfLine);
    System.out.print('*');
    spaces(insideWidth);
    System.out.println('*');
    insideWidth = insideWidth + 2;
    startOfLine--;//THIS LINE WILL MOVE.
}
```

drawTop方法的完整定义参见图7-12。上述的循环在其中被高亮突出显示。此外，为了配合循环之前的代码，下面的代码被移到循环开头而不是放在结尾处，不过仍旧保持每次循环迭代减少1。

```
startOfLine--;
```

完成本工程时，所编写的代码应当和图7-13中列出的示例应用程序相同。

```
public class GraphicsDemo
{
    public static final int indent = 5;
    public static final int topWidth = 21;
    public static final int bottomWidth = 4;
    public static final int bottomHeight = 4;
    public static void main(String[] args)
    {
        System.out.println("            Save the Redwoods!");
        Triangle top = new Triangle(indent, topWidth);
        Box base = new Box(indent + (topWidth/2) - (bottomWidth/2),
                                    bottomHeight, bottomWidth);
        top.drawAt(1);
        base.drawAt(0);
    }
}
```

屏幕输出

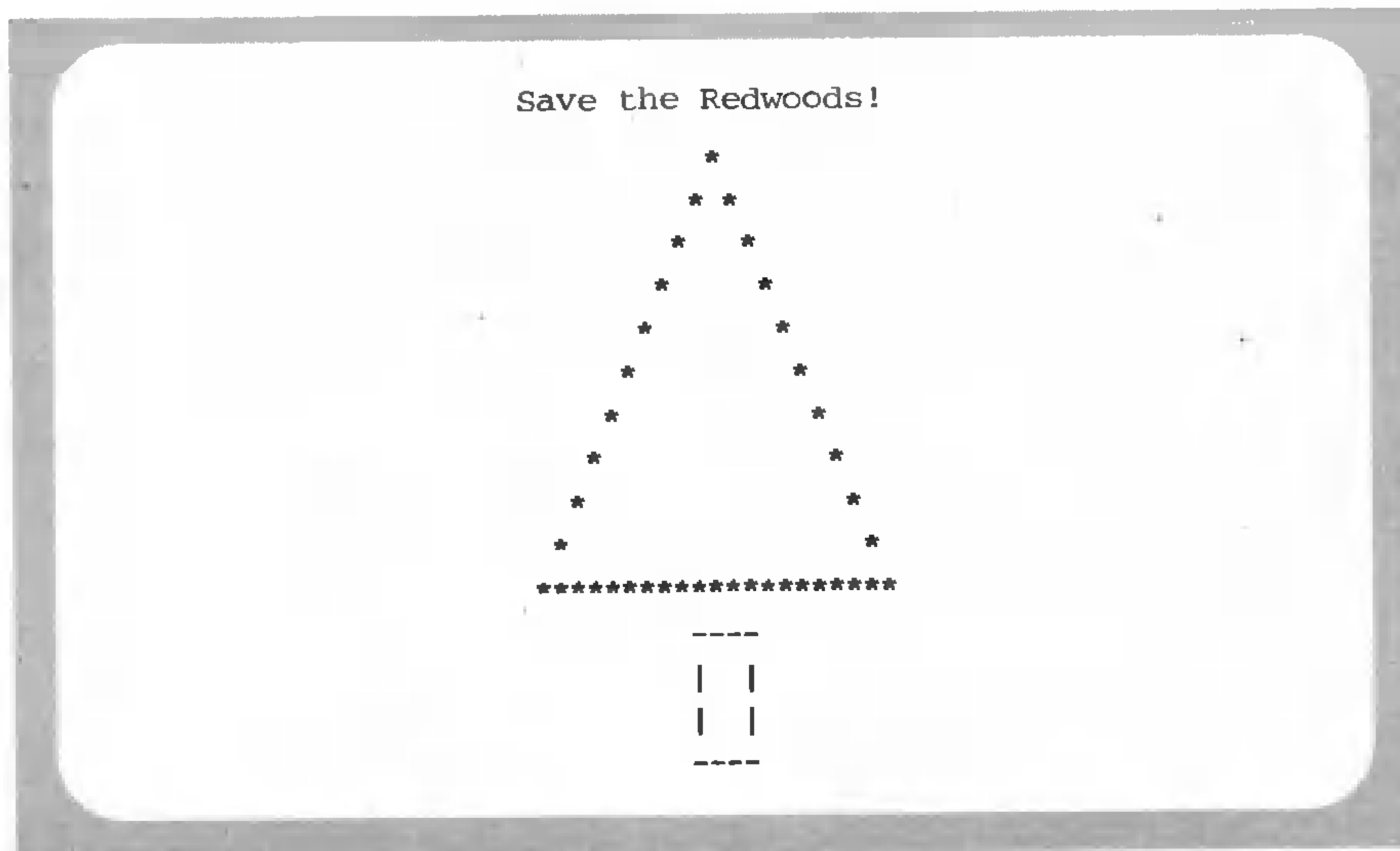


图7-13 字符图形应用程序

自测题

16. 定义一个名为Diamond的类作为Figure类的派生类。Diamond类和Triangle类差不多，不过，画Diamond时，上半部分和Triangle相同，而下半部分是上半部分的倒置。

7.2.6 抽象类

图7-10中定义的Figure类不是为了创建Figure类的对象用的，而是设计作为基类用来派生其他类的，比如Box类（图7-11）。尽管不需要创建Figure类的对象，但这样做仍旧是合法的，例如：

```
Figure figureVariable;
figureVariable = new Figure();
```

不过，为了使之合法，需要在Figure类中给出drawHere方法的定义。Figure类中的drawHere方法的定义是个摆设，它画了一个星号，只是使调用它的时候有点儿事做。我们不会用Figure对象来调用其drawHere方法，而是想对派生类调用drawHere方法（比如派生类Box和Triangle）。

在与此类似的情况下，若有方法希望被派生类重写而且只被派生类的对象调用，则可以把该方法声明为**抽象的**（abstract）。例如：

```
public abstract void drawHere();
```

注意，方法头部中用了关键字abstract，而且方法头部之后紧跟一个分号，没有方法体。抽象方法不会被任何对象使用，在（非抽象的）派生类中它必须被重写，以给出一个“新”的定义。

Java要求，如果类有至少一个抽象方法，这个类就必须被声明为抽象的。具体做法就是在类定义的头部加关键字abstract，示例代码如下：

```
public abstract class Figure
{
    ...
```

这样定义的类被称为**抽象类**（abstract class）。 如果某类是抽象的，就不会存在这类的具体实例对象（除非是其派生类的）。抽象类只能作为基类来定义派生类。

在图7-14中，Figure类被重新修订为抽象类。如果用这个抽象版本的Figure类，它的派生类都会和以往一样工作，只是我们不能创建仅是普通Figure类的对象了。

```
/**
Abstract class for simple character graphics figures to send to the screen. It is
intended to be used as a base class for the kinds of figures that will be used in
graphics applications.
*/
public abstract class Figure
{
    private int offset;
    public abstract void drawHere();
```

除了drawHere之外，所有的构造器和方法与图7-10中的相同。除了drawHere，其他方法的首部都没有用abstract关键字。我们把其中一个方法复制在下面。

```
    /**
    Draws the figure at lineNumber lines down from the current line.
    */
    public void drawAt(int lineNumber)
    {
        int count;
        for (count = 0; count < lineNumber; count++)
            System.out.println();
        drawHere();
    }
}
```

图7-14　重新修订为抽象类的Figure

尽管Figure类是抽象的，也并非其所有方法都是抽象的。除了drawHere方法之外，其他

方法仍然和以前的定义一样，它们是完整的定义（并且没有使用abstract关键字）。如果在抽象类中给出一个通常的方法定义是有意义的，就应当这么做。这样，可以将尽可能多的细节放在抽象类中，每个派生类就不必重复这些细节了。

为什么要使用抽象类？因为它能简化思路。我们已经解释过定义Figure类使我们可以对各种图形只需要定义一次drawAt方法。抽象类使定义Figure这样的类更容易，因为不需要去写没什么用处的方法定义了。如果因为方法总是会被重写而使这个方法的定义变得没有意义，那就把它设为抽象方法（并且把这个类也抽象化），这样就不需要写这个无意义的方法定义了。

尽管抽象的方法没有一个完整的定义，但它还是有用的。它是方法的占位符，该方法必须在所有的（非抽象的）派生类中被定义。图7-14所示的drawAt方法中有对drawHere方法的调用。如果省略了抽象方法drawHere，对drawHere的调用就不合法了。

自测题

17. 如果Figure如图7-14所示定义，下面的代码还合法吗？

```
Figure figureVariable = new Figure();
```

18. 如果Figure是按照图7-14所示定义，而Box是按照图7-11所示定义，下面的代码合法吗？

```
Figure figureVariable = new Box();
```

记住：抽象类是类型

对于图7-14中所示的Figure那样的抽象类，不能创建其对象，即对图7-14中的Figure版本来说，下面的代码不合法：

```
Figure f = new Figure();
```

尽管如此，形参的类型是Figure还是很合理的。这样，Figure类的任何子孙类的对象都可以代入这个参数。

7.2.7 接口

接口（interface）可以算是抽象类的极端形式。实际上，它太极端了，以至于接口不是一个类。不过，它是一个类型，任何实现了*该接口*的类都具有该类型。

接口定义了一些方法的头部，任何类要实现该接口，就必须定义这些方法。例如，图7-15列出了一个名为Writable的接口。接口中只有方法的头部，不包含实例变量或者完整的方法定义。（不过接口可以包含定义好的常量。）

```
public interface Writable
{
    public String toString();
    public void writeOutput();
}
```

别忘了方法首部结尾的分号。

图7-15 Writable接口（选读）

要实现一个接口，类必须做到下面两件事：

（1）必须将下面的子句置于类定义的开头。

```
implements Interface_Name
```

要实现多个接口，只需列出所有接口的名字，用逗号分隔，例如：

```
implements MyInterface, YourInterface
```

（2）必须实现接口的定义中列出的所有方法。

例如，要实现Writable接口，类定义必须在开头包含子句implements Writable，就像下面的代码这样：

```
public class WritableUndergraduate extends Student implements Writable
{
```

该类还必须实现writeOutput()和toString()方法。WritableUndergraduate的完整定义参见图7-16。

```
public class WritableUndergraduate extends Student implements Writable
{
    private int level; //1 for freshman, 2 for sophomore,
                       //3 for junior, or 4 for senior.

    public WritableUndergraduate()
    {
        super();
        level = 1;
    }

    public WritableUndergraduate(String initialName,
                    int initialStudentNumber, int initialLevel)
    {
        super(initialName, initialStudentNumber);
        setLevel(initialLevel); //Checks 1 <= initialLevel <= 4
    }

    public String toString()
    {
        return ("Name: " + getName() +
            "\nStudent number: " + getStudentNumber() +
            "\nLevel: "+ getLevel());
    }

    public void writeOutput()
    {
        super.writeOutput();
        System.out.println("Student Level: " + level);
    }

    public boolean equals(WritableUndergraduate other)
    {
        return (super.equals(other)
            && (this.level == other.level));
    }
    <其他方法的定义与图7-7中的Undergraduate类相同。>
}
```

图7-16　实现一个接口

接口是一个类型。方法可以用接口类型作为参数，如Writable类型，这样，方法就可以作用于其后定义的、实现该接口的类了。接口起的作用类似于基类，但是必须强调的是它不是基类。（实际上，它不是任何形式的类。）有些程序设计语言允许一个类是两个不同基类的派生类，但在Java中这是不允许的。Java里的派生类只能有一个基类。但是，在基类之外，Java类可以实现任意多的接口。这使Java程序能有近似于多重基类的能力，但是避免了多重基类带来的复杂性。

接口的定义存储在一个.java文件中，同类定义一样被编译。

7.3 动态绑定和多态

本节介绍可以让不同的对象对同一个方法名（如drawAt）有不同的定义的一种途径。例如b是一个Box类型的对象，t是一个Triangle类型的对象，则b和t用的就是不同的drawAt定义，哪怕b和t可能都被同类型的变量所命名——在我们的例子中，两个对象都被Figure类型的变量命名。

7.3.1 动态绑定

图7-10中Figure类的drawAt方法的定义对drawHere方法进行调用。如果只有Figure一个类，那就没什么特别。但是我们从基类Figure派生出了Box类。Box类从Figure类一成不变地继承了drawAt方法，但是Box类重写了drawHere方法的定义。“那会怎么样？”读者可能会问。千变万化！例如，对于下面的代码，编译器该做些什么：

```
Box b = new Box(1, 4, 4);
b.drawAt(2);
```

drawAt方法在Figure中定义，但是它调用了在Box类中重定义的drawHere方法。drawAt方法的代码同Figure类一同编译，并且这个类在Box类及其drawHere方法写好之前就编译好了。因此，已经编译好的drawAt方法所使用的drawHere方法的定义，在编译drawAt时，甚至都还没有写好，这是如何做到的呢？

编译drawAt的代码时，在调用drawHere的地方插入的不是代码，而是一个注释：“使用当前可用的drawHere的定义。”此后，当调用b.drawAt(2)时，编译的drawAt代码最终会遇到这个相当于说“使用当前可用的drawHere的定义。”的注释。这个注释被替换为对b的drawHere版本的调用。如果b是Box类型，则这里用的drawHere就是Box类定义中的版本。

这种对调用以后可能被重写的方法的处理叫作**动态绑定**（dynamic binding）或**推迟绑定**（late binding），因为在运行程序之前，方法调用的意义没有和方法调用的位置绑定。如果Java不用动态绑定，则运行f.drawAt()时，总是会看到Figure类的drawAt方法的效果（会是一个单独的星号），甚至当f是Box或是Triangle时也如此。

其他程序设计语言并不一定像Java这样自动进行动态绑定。在很多其他程序设计语言中，必须进一步指明哪些方法需要动态绑定。Java的处理方式效率稍低但是意味着Java语言更易于进行编程而且不太容易出错。

为了看出动态绑定确实重要，考察下面的代码，记住drawAt中有对drawHere的调用：

```
Figure f;
Box b = new Box(1, 4, 4);
```

```
f = b;
f.drawAt(2);
Triangle t = new Triangle(1, 21);
f = t;
f.drawAt(2);
```

用颜色标明的两行是相同的，而且drawAt方法的代码也是相同的，因为两者都从Figure继承了这个方法的定义。然而它们用了不同的drawHere方法定义，而这会被drawAt调用，因此两处对drawAt的调用产生不同的输出。第一个画了盒子而第二个画了三角形。对象在被用new创建时就记住了它有些什么方法定义。

Box和Triangle类从Figure类继承了drawAt方法。drawAt方法在Box和Triangle类中都没有被重写。因此Box和Triangle对象的drawAt方法的实现文本是一样的。在drawAt的定义中调用的drawHere方法被（直接）重写了。在此情况下，可以说方法名drawAt在Box和Triangle类中被**间接重写**（**indirectly overridden**）了。

注意到哪个方法定义会被使用取决于对象在继承链中的位置，这一点很重要。它不是由命名这个对象的变量的类型决定的。例如：

```
Box b = new Box(1, 4, 4);
Figure f = b;
f.drawHere();
```

把Box类的对象赋值给Figure类型的变量毫无问题。但是，该对象还是记住了它是作为一个Box被创建的。在此情况下，f.drawHere()将使用Box中给出的drawHere的定义，而不是Figure中给出的drawHere的定义。为了确定哪个drawHere的定义将被使用，Java检查该对象用new创建时使用的是什么类①。

记住：对象自己清楚该如何去做

当调用被重写的方法（或者使用被重写的方法的方法）时，该方法的行为是由（用new）创建该对象的类定义的。它不是由命名这个对象的变量的类型决定的。任何祖先类的变量都可以持有子孙类的对象，但是对象始终记住了每个方法名对应的方法行为。变量的类型不起作用。起作用的是该对象被创建时类的名字。这是因为Java使用的是动态绑定。

7.3.2 类型检查与动态绑定

你需要清楚动态绑定如何与Java编译器的类型检查相互作用。例如，如果Employee是Person类的派生类，可以将Employee类型的对象赋值给Person类型的变量，例如：

```
Employee e = new Employee()
Person p;
p = e;
```

---

① Java很善于发现应当使用哪个方法定义，甚至强制类型转换也欺骗不了它。b.drawHere()的意义始终是由Box类中的方法定义的，即便用强制类型转换把b的类型改为Figure，例如：

```
Box b = new Box(1, 4, 4);
Figure f = (Figure)b;
f.drawHere();
```

在这里，f.drawHere()将使用Box类中定义的drawHere，而不是Figure类中定义的drawHere。

但是事情还没有完。

尽管可以把Employee类型的对象赋值给Person类型的变量，但是当调用对象是变量p时（类型为Person），只能调用在Person类中有的方法。不过，如果该方法在Employee类的定义中被重写，而且变量p命名的对象的类型是Employee，则会使用Employee中定义的版本。因此变量决定了哪些方法名可以使用，但是对象决定了会使用这些方法名的哪个版本。如果要对Person类型的变量p命名的对象使用一个在Employee类中首次引进的方法，就必须使用强制类型转换。下面的代码就可以工作（假设setEmployeeNumber首次在Employee类中定义）：

```
Employee e = (Employee)p;
e.setEmployeeNumber(5678);
```

另一个例子也会有帮助。图7-3的Student类是图7-1中Person类的派生类。现在假设在程序中有如下代码：

```
Person p = new Student("Joe", 1234);
```

那么下面的代码就是合法的：

```
p.setName("Josephine");
p.writeOutput();
```

第二个调用将使用Student类中的writeOutput的定义。（*是对象，而不是变量，决定了哪个方法的定义会被使用。*）

另一方面，下面的代码是不合法的，因为setStudentNumber不是Person类的方法名。（变量决定了哪些方法名可以被使用。）

```
p.setStudentNumber(1234); //ILLEGAL
```

变量p是Person类型，但是变量p中的对象仍旧是Student类型的对象。因此，尽管该对象仍然可以调用setStudentNumber方法，但是编译器并不知道！为了使之合法，需要进行强制类型转换，代码如下：

```
Student s = (Student)p;
s.setStudentNumber(1234); //Legal
```

记住：变量决定了可以使用哪些方法名，但对象决定了使用该方法名的哪个定义

当对象被用new按照某个类创建，但是存储在一个祖先类的变量中时，变量决定了该变量名可以使用哪些方法，然而用于创建该对象的类决定了可以使用该方法名的哪个定义。

你可能会认为这只不过是个愚蠢的练习，因为不会将Student类型的对象赋值给Person类型的变量，但并非如此。可能不会经常直接进行这样的赋值，但是可能会经常无意识地这么做。回想一下，可以把一个Student类型的参数“代入”，作为一个Person类型的方法参数。而参数是个局部变量，被赋为“代入”的实参的值。在这种情况下，Student类型的对象（方法调用时的实参）就被赋值给Person 类型的变量（方法定义中的参数）。是的，实际上是对象的引用（即对象的内存地址）被赋值给变量，但这是个次要问题而与当前的问题无关。

7.3.3 对toString的动态绑定

在7.2.5节中注意到，如果给Student类添加一个合适的toString方法，就可以用

toString方法向屏幕输出，例如：

```
Student s = new Student("Joe Student", 2001);
System.out.println(s.toString());
```

得益于动态绑定，甚至在调用System.out.println时连toString都不需要使用。下面的代码也能工作良好，产生完全相同的输出：

```
Student s = new Student("Joe Student", 2001);
System.out.println(s);
```

System.out.println(s)方法调用是在对象System.out上调用println方法。其中一个println方法的定义是这样的，它有单独一个Object类型的实参。该定义和下面的代码等价：

```
public void println(Object theObject)
{
    System.out.println(theObject.toString());
}
```

(在括号中调用的println方法是一个不同的，重载的println方法。括号中调用的println方法有一个String类型的参数，不是Object类型的参数。)

这个println的定义在Student类被定义之前就已经有了，但是调用：

```
System.out.println(s);
```

用一个Student类型（因此也是Object类型）的实参s，所以使用的是Student类中的toString定义，而不是Object类中的toString定义。让它们这样工作的正是动态绑定。

7.3.4 多态

多态（polymorphism）这个词来自一个希腊语单词，意思是“很多形态”。在计算机科学中，它的原始意义是指根据上下文，有能力使用同一个方法名的不同定义。对于这个原始意义来说，像重载这样的行为也可以认为是多态。但是，这个术语如今更加专用。现在，它指的是动态绑定机制，用来决定会使用被（直接或间接）重写的方法名的哪个方法的行为。这样，在现在的用法中，多态基本上是动态绑定的同义词。

那么为什么本节的标题是“动态绑定与多态”？这难道不是在说“动态绑定与动态绑定”吗？确实不错，但是这两个术语仍然有些不同。动态绑定被认为是计算机采取的一个处理过程，而多态被认为是对象的某种行为。

多态在面向对象程序设计中扮演了一个重要的角色。实际上，大多数程序员认为封装、继承及多态是面向对象程序设计的主要特征。因此，多态是面向对象程序设计哲学的3个核心特征之一。

快速参考：多态

多态意味着运用动态绑定过程允许不同的对象对同一个方法名采用不同的方法定义。

■ Java提示：更好的equals方法(选读)

在本章前面讨论Object类时，提到它有一个equals方法，以及当定义一个有equals方法的类时，应当重写Object类中的equals方法定义。但是，严格来说，我们没有按照自己的建议那样去做。

在图7-3定义的Student类中，方法equals的头部如下：

```
public boolean equals(Student otherStudent)
```

而Object类中equals方法的头部如下：

```
public boolean equals(Object otherObject)
```

注意，两个equals方法有不同的参数类型，因此我们并没有重写equals的定义，只是重载了equals方法。Student类有这两个名为equals的方法。

大多数情况下，这没什么大不了的。但是，有些情况下这确实会带来问题。假设用某个预定义的方法（或程序员自定义的方法），它有一个名为objectPar的参数的类型是Object，而另一个名为studentPar的参数的类型是Student。现在假设该方法的代码中有如下语句：

```
studentPar.equals(objectPar)
```

如果用两个Student类型的实参代入参数objectPar和studentPar，Java将使用Object类中定义的equals，而不是Student类中定义的equals。这意味着在某些情况下，equals方法会返回错误的答案。

为了解决这个问题，需要改变Student类中的equals方法的参数类型，从Student改为Object。例如：

```
public boolean equals(Object otherObject)
{
    Student otherStudent = (Student)otherObject;
    return (this.sameName(otherStudent)
        && (this.studentNumber == otherStudent.studentNumber));
}
```

注意，需要把参数otherObject从Object类型转换为Student类型。如果省略这个转换，编译器将在看到下面的表达式时给出错误信息：

```
otherObject.studentNumber
```

因为Object类没有名为studentNumber的实例变量。（实际上，很可能在此之前就给出错误信息了，它可能会抱怨无法调用sameName。）

改进equals的初步尝试确实重写了Object类中的equals定义，而且在大多数情况下工作良好，但是它还有个缺点。

equals定义现在允许实参是任何对象。如果使用equals方法时给出的实参不是Student，将会怎样？答案是当强制转换为Student类型时，会发生运行时错误。

应当让我们的定义对任何对象都可以工作。如果对象不是Student类型，可以很简单地返回false。（调用对象是Student类型，因此如果实参不是Student类型，不应该认为它们是相等的。）但是如何才能知道实参是不是Student类型呢？

instanceof操作符可以用于检查某个对象是不是Student类型。语法如下：

Object instanceof *Class_Name*

如果*Object*是*Class_Name*类型，这个表达式返回true，否则就返回false。因此如果otherObject是Student类型，下面的表达式会返回true：

```
(otherObject instanceof Student)
```

如果上述表达式不是true，equals方法应当返回false。

我们的equals方法的最终版本见图7-17。注意还对另一个可能的情况进行了仔细处理。预定义的常量null可以用于代入类型为Object的参数。Java文档中说明equals方法应当在将对象和null进行比较时返回false。我们正是这么做的。

```
public boolean equals(Object otherObject)
{
    if (otherObject == null)
        return false;
    else if (!(otherObject instanceof Student))
        return false;
    else
    {
        Student otherStudent = (Student)otherObject;
        return (this.sameName(otherStudent)
            && (this.studentNumber == otherStudent.studentNumber));
    }
}
```

图7-17 Student类的更好的equals方法(选读)

自测题

19. 重写一个方法名与重载一个方法名有何不同?
20. 图7-11中的drawHere方法定义是重载的例子还是重写的例子?
21. 图7-12中的drawHere方法定义是重载的例子还是重写的例子?
22. 图7-12中的两个构造器的定义是重载的例子还是重写的例子?
23. 什么是多态?
24. 什么是动态绑定? 什么是推迟绑定? 请给出各自的例子。
25. 重载一个方法名是不是多态的例子?
26. 在7.3.4节中描述的面向对象程序设计的3个主要特征是什么?
27. 下面的代码中，两次对drawAt的调用在屏幕上的输出图形是否相同? (这些类的定义依照前文对应的节。)

```
Figure f;
f = new Box(1, 4, 4);
f.drawAt(2);
f = new Triangle(1, 21);
f.drawAt(2);
```

7.4 图形编程补充（选读）

我的天哪! 在不知不觉中我已经说了40多年散文竟然还不知道。

——莫里哀，法国剧作家，《飞上枝头作名流》

本节要讲解在每次写applet的定义时是如何使用继承的。还将讲解如何写出视窗界面，使其运行效果像通常的Java应用程序，而不是applet。

7.4.1 JApplet类

你应当可以从我们为那些applet类所写的代码中看出applet是JApplet类的派生类，这从applet类的定义的第一行就可以看出。例如：

```
public class LabelDemo extends JApplet
```

JApplet类有init和paint方法。因此，定义一个applet的init或paint方法时，就是在重写继承来的init或paint方法。这就是init和paint方法可以被自动调用的原因。其细节大致如下：假设有个方法名为showApplet，它有一个名为anApplet的JApplet[①]类型的参数。showApplet方法可以在其中调用anApplet.init()以及（或）anApplet.paint()，因为JApplet类有名为init()和paint()的方法。得益于多态，只要用你的applet类代入参数anApplet，对showApplet的调用就会调用到你在自己的applet中定义的init()和paint()。这样，在你写好自己的applet之前，库中那些定义好的类就可以调用你的init()和paint()方法了。这些库中的类方法可以运行你的applet并自动调用你定义的init()和paint()。

7.4.2 JFrame类

图形用户界面（Graphical User Interface，GUI）其实就是某些类型的程序的视窗界面。applet是一种GUI。JApplet类作为基类来派生出applet，以在Web页面上运行（尽管到目前为止我们只是在applet查看程序中运行它们）。为了能获得GUI（视窗界面）以便可以作为普通的Java应用程序来运行，就要使用JFrame（而不是JApplet）作为基类。在图7-18里，重写了图5-26中的applet：把基类JApplet替换为JFrame，并做了一些必需的调整来适应这个改变。改变之后的类会产生一个不再需要引用Web页面的GUI。不过，为了“运行”这个GUI，还需要一个Java应用程序创建出图7-18中的这个ButtonDemo类的对象。在图7-19里，编写了一个简单的应用程序，创建出一个名为gui的对象，类型为ButtonDemo，用下面的语句显示出这个GUI：

```
gui.setVisible(true);
```

JFrame类，以及从JFrame派生的每一个类，都有一个名为setVisible的方法。该方法接受一个boolean类型的实参。若该实参为true，就可以让GUI可见；如果该实参为false，就隐藏GUI。这和5.8节中讨论的setVisible方法是一样的。标签、按钮、JFrame以及其他组件最终都从一个公有的祖先类继承了这个setVisible方法。正是继承性赋予了setVisible方法在如此广泛的类中都有一致的行为。

回到图7-18中的从JFrame派生的GUI类，解释其中与相关的图5-28中的applet代码不同的部分。有差别的地方都高亮突出显示。最显著的不同是extends JApplet被替换为extends JFrame。

另一个主要的差异是从JFrame派生的类没有init方法，但是用了一个构造器。本该放在applet的init方法中的代码被放在了这个从JFrame继承的类的构造器中。从某种意义上说，这不需要解释。它们都是初始化动作，而且初始化的动作是属于构造器管辖的。实际上，使用init方法比使用构造器更加反常。

其他的差异就是添加到这个从JFrame派生出来的类中的各种内容，它们在图5-28的applet类中不曾出现。下面就来讨论这些新增部分。

从JFrame派生出的GUI调用setSize方法来设置自己的初始尺寸，下面的代码摘自图7-18：

① 方法名showApplet是虚构的，代表任意的有一个JApplet类型的参数的方法。

```
setSize(WIDTH, HEIGHT);
```

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

/**
 Simple demonstration of putting buttons in a JFrame.
*/
public class ButtonDemo extends JFrame implements ActionListener
{
    public static final int WIDTH = 400;
    public static final int HEIGHT = 300;

    public ButtonDemo()
    {
        setSize(WIDTH, HEIGHT);
        WindowDestroyer listener = new WindowDestroyer();
        addWindowListener(listener);

        Container contentPane = getContentPane();
        contentPane.setBackground(Color.WHITE);

        contentPane.setLayout(new FlowLayout());

        JButton sunnyButton = new JButton("Sunny");
        sunnyButton.addActionListener(this);
        contentPane.add(sunnyButton);
        JButton cloudyButton = new JButton("Cloudy");
        cloudyButton.addActionListener(this);
        contentPane.add(cloudyButton);
    }

    public void actionPerformed(ActionEvent e)
    {
        String actionCommand = e.getActionCommand();
        Container contentPane = getContentPane();

        if (actionCommand.equals("Sunny"))
            contentPane.setBackground(Color.BLUE);
        else if (actionCommand.equals("Cloudy"))
            contentPane.setBackground(Color.GRAY);
        else
            System.out.println("Error in button interface.");
    }
}
```

为了编译这个类，必须要有WindowDestroyer类在该类的同一个目录中。在本节稍后部分会讨论WindowDestroyer类。

图7-18　从JFrame派生出的视窗界面

int类型的值给出宽度和高度，单位是像素。setSize方法对applet也有效，但是更常见的是在显示这个applet的Web页面上用代码来设置其尺寸。applet查看程序会把applet的尺寸设置为一个默认值，因此尽管合法，但是一般不会对applet使用setSize方法。另一方面，如果不对从JFrame派生出的GUI使用setSize，这个GUI就可能会太小了。

当显示applet的Web页面关闭，或者点击applet查看程序的关闭窗口按钮，applet就会被终止（你已经见到的那个在applet上的关闭窗口按钮不是applet自己的关闭窗口按钮，而是属于

查看程序的）。使用JFrame时，需要对这个GUI上的关闭窗口按钮编程。7.4.3节会讨论这个问题。

```
public class ShowButtonDemo
{
    public static void main(String[] args)
    {
        ButtonDemo gui = new ButtonDemo();
        gui.setVisible(true);
    }
}
```

得到的GUI



这个GUI会响应按钮点击而改变颜色，表现和图5-26中的applet相同。

图7-19 在应用程序中运行JFrame类

快速参考：setSize方法

setSize方法把JFrame对象的尺寸重新设置为指定的宽度和高度。

语法：

JFramObject.setSize(*Width, Height*);

举例：

this.setSize(400, 300);

setSize方法最常见的用法是在派生自JFrame的类的构造器中调用，此时this一般可以省略。因此，更常见的示例如下：

setSize(400, 300);

常见问题：宽度和高度的单位是什么？

使用setSize时，宽度和高度实参是按照像素数给出的。在1.4节中讨论过像素。

7.4.3 窗口事件与窗口侦听器

需要对从JFrame派生出的GUI的关闭窗口按钮编程。关闭窗口按钮会发出一个事件，就像曾见到过的普通按钮（JButton）。但不同之处是，关闭窗口按钮产生一个**窗口事件**（window event），被**窗口侦听器**（window listener）捕获。

WindowAdapter类是一个窗口侦听器，因此每个从WindowAdapter派生的类也是窗口侦听器，如图7-20中给出的WindowDestroyer类。使用addWindowListener方法可以把窗口侦

听器注册到JFrame的GUI（这与对JButton使用addActionListner完全类似）。下面的代码创建一个WindowDestroyer对象并将其注册到图7-18中的JFrame GUI：

```
WindowDestroyer listener = new WindowDestroyer();
addWindowListener(listener);
```

WindowDestroyer类不是标准Java库中的类。你需要自行定义它。不过，正如图7-20里列出的定义，这个WindowDestroyer类很简单。windowClosing方法对关闭窗口按钮的响应方式和actionPerformed响应JButton事件的方式很相像。因此，当关闭窗口按钮按下时，windowClosing方法调用了System.exit，随后JFrame GUI会终止。在本书姊妹篇《Java程序设计与问题解决：高级篇（第4版）》中会更充分地讨论窗口侦听器以及WindowAdapter和WindowDestroyer类。在此之前，你可以简单地从图7-18中复制WindowDestroyer的定义，并总是在从JFrame派生出的GUI的构造器中包含如下代码：

```
WindowDestroyer listener = new WindowDestroyer();
addWindowListener(listener);
```

```
import java.awt.*;
import java.awt.event.*;

/**
 If you register an object of this class as a listener to any object of the class
 JFrame, then if the user clicks the close-window button in the JFrame, the object of
 this class will end the program and close the JFrame.
*/
public class WindowDestroyer extends WindowAdapter
{
    public void windowClosing(WindowEvent e)
    {
        System.exit(0);
    }
}
```

Java（标准库）中没有预先定义好这个WindowDestroyer类，程序员需要自行定义它。

图7-20 响应来自JFrame GUI的窗口事件的侦听器类

▲ 易犯错误：不要混淆JButton和关闭窗口按钮

JFrame GUI的关闭窗口按钮不是JButton。这个按钮发出窗口事件，由窗口侦听器（比如WindowDestroyer）捕获。JButton发出动作事件，由动作侦听器捕获。JFrame GUI总是有一个关闭窗口按钮，并且可能有JButton。 △

7.4.4 ActionListener接口

在7.2节介绍了接口，不过如果你已经阅读了前面几章图形编程补充的章节，那么从第5章起就用过一个特殊的接口了，名为ActionListener。

ActionListener接口只有一个方法头部，这是一个类要实现这个接口所必需要具备的：

```
public void actionPerformed(ActionEvent e)
```

响应按钮点击动作的侦听器（不管是在applet中还是在JFrame中）必须是一个**动作侦听**

器（action listener），这只是说一个按钮的侦听器必须实现该ActionListener接口而已。

编程示例：用JFrame实现的笑脸

JFrame类及所有从JFrame派生的类都有一个paint方法。和applet一样，可以重写JFrame GUI的paint方法来画图。在图7-21中，我们重写了图1-8的applet，得到一个显示开心笑脸的JFrame GUI。

图7-21中JFrame GUI的paint方法与图1-6中的paint方法相同，对它的代码的解释也与图1-6中的paint方法相同。

```
import javax.swing.*;
import java.awt.*;
public class HappyFace extends JFrame
{
    public static final int WIDTH = 400;
    public static final int HEIGHT = 300;

    public HappyFace()
    {
        setSize(WIDTH, HEIGHT);
        WindowDestroyer listener = new WindowDestroyer();
        addWindowListener(listener);
    }

    public void paint (Graphics canvas)
    {
        canvas.drawOval(100, 70, 200, 200);
        canvas.fillOval(155, 120, 10, 20);
        canvas.fillOval(230, 120, 10, 20);
        canvas.drawArc(150, 195, 100, 50, 180, 180);
    }
}
```

图7-21 在JFrame中画图

运行JFrame，显示笑脸，如图7-22所示。

```
public class ShowHappyFace
{
    public static void main(String[] args)
    {
        HappyFace gui = new HappyFace();
        gui.setVisible(true);
    }
}
```

得到的GUI

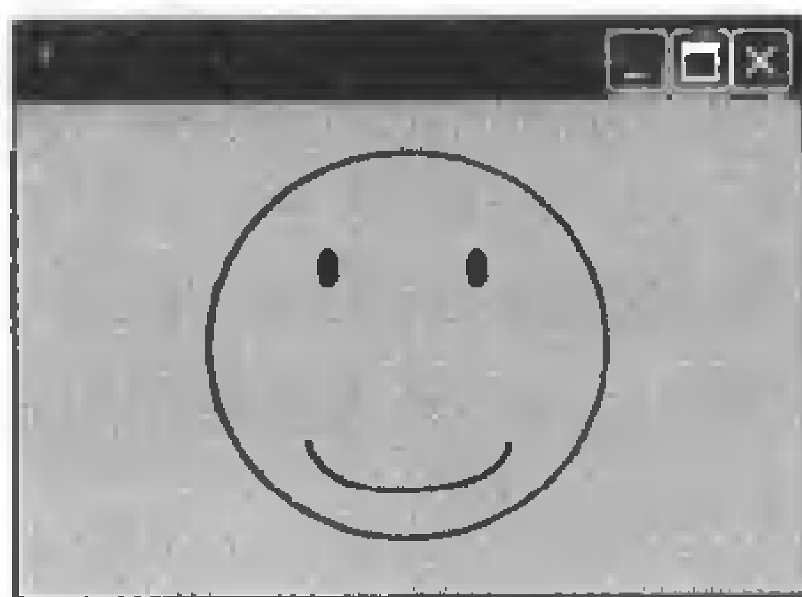


图7-22 运行JFrame，显示出所绘图形

自测题

28. WindowDestroyer是否在任何标准Java包（库）中？
29. 在applet中的paint方法和从JFrame派生出的GUI中paint的方法里面，能做的事情有何不同？
30. 通常从JFrame派生出的GUI没有init方法。在这样的GUI里，原先放在applet的init方法中的代码应放在哪里？

7.4.5 下一步该如何阅读

接下去可以按顺序阅读本书姊妹篇《Java程序设计与问题解决：高级篇（第4版）》，不过，如果想更早地全面了解GUI，可以先阅读该书第5章。

小　　结

- 向基类中添加额外的实例变量或方法能构造派生类。派生类继承了基类的全部实例变量和方法。
- 为派生类定义构造器时，其定义需要首先调用基类的构造器。如果没有进行显式调用，Java则会自动调用基类的默认构造器。
- 可以重新定义来自基类的方法，使之在派生类中具有不同定义。这种做法称为重写方法定义。
- 重写方法定义时，在派生类中给出的新定义与以前的定义有同样数目和类型的参数。如果派生类中的方法与基类中的方法的参数数目不同，或者相同位置的参数类型不同，则是重载。
- 基类中的私有实例变量和私有方法不能在派生类中直接访问。
- 假设A是类B的派生类，那么一个类A的对象不但是类A的对象（当然如此），同时也是类B的对象。
- 多态是指采用动态绑定来使不同的对象对于同样的方法名具有不同的方法行为。
- 从JFrame派生的类产生出类似applet的视窗界面，但它运行在一个普通的Java应用程序中，而不是运行在Web页面或者查看程序中。①
- 从WindowAdapter派生的类是一种窗口侦听器，因此可以响应JFrame中的关闭窗口按钮的点击。①

✓ 自测题答案

1. 正确，派生类拥有基类的全部实例变量，并且可以添加更多的实例变量。
2. 正确，它有这个方法。派生类拥有基类的全部公有方法（并且可以添加更多的方法）。如果派生类没有重定义（重写）方法的定义，那么它在派生类中的行为和在基类中完全相同。不过，派生类可以给方法一个新的（重写的）定义，这个新定义会取代原定义（两者的参数数目和类型必须相同）。
3. 不能。
4. 不能。
5. sameName方法没有在Student的类图中列出。于是沿着箭头来到Person类图。Person类图中名为sameName的方法具有单个类型为Person的参数。因为我们知道Student是一个Person，那么这个定义适合于题中这个参数类型是Student的sameName方法。因此该方法使用的定义在Person类的定义中。
6. 从Student的类图开始。Student类图中的setStudentNumber方法有单个类型为int的参数，因此不需要进一步寻找了。题中的setStudentNumber方法使用的定义就在Student类中。
7.
```
public class TitledPerson extends Person
{
```

① 7.4节介绍的内容。

```
    private String title;

    public TitledPerson()
    {
        super();
        title = "no title yet";
    }

    public TitledPerson(String initialName, String initialTitle)
    {
        super(initialName);
        title = initialTitle;
    }

    public void reset(String newName, String newTitle)
    {
        setName(newName);
        title = newTitle;
    }

    public String getTitle()
    {
        return title;
    }

    public void setTitle(String newTitle)
    {
        title = newTitle;
    }

    public void writeOutput()
    {
        System.out.println("Name: " + getName());
        System.out.println("Title: " + title);
    }

    public boolean equals(TitledPerson otherPerson)
    {
        return (this.sameName(otherPerson)) &&
               (this.title.equalsIgnoreCase(otherPerson.title));
    }
}
```

8.
```
public void writeOutput()
{
    System.out.println("Name: " + getName());
    System.out.println("Student Number: "+ getStudentNumber());
    System.out.println("Student Level: " + level);
}
```

9.
```
public void reset(String newName, int newStudentNumber, int newLevel)
{
    setName(newName);
```

```
        setStudentNumber(newStudentNumber);
        level = newLevel;
    }
```

10. 是的，一个对象可以有多种类型。假设类A从类B派生而来，则类A的对象是A类型也是B类型。
11. 该对象有下列类型：Undergraduate、Student及Person。(你将很快在本章中发现，它还有Object类型，这是Java中预定义的类。)
12. 首先它可以用作基类的构造器名（见图7-3)。当下面的代码用作“发起调用的对象”时：

```
super.writeOutput();
```

它被用于表明应该使用基类中指定名称的方法。例如，在图7-7中，它说明要使用基类Student的writeOutput方法，而不是Undergraduate类的writeOutput方法。
13. 关键字this用作方法名时，是指该类的构造器。关键字super用作方法名时，是指派生类的基类的构造器。
14. 因为toString 的返回值中包含了换行字符'\n'。
15.

```
Person p1 = new Student(); //Valid. A Student is a Person.
Person p2 = new Undergraduate(); //Valid.
                           //An Undergraduate is a Person.
Student s1 = new Person(); //ILLEGAL!
Student s2 = new Undergraduate(); //Valid.
                           //An Undergraduate is a Student.
Undergraduate ug1 = new Person(); //ILLEGAL!
Undergraduate ug2 = new Student(); //ILLEGAL!
Object ob = new Student(); //Valid.
                           //A Student is an Object.
Student s3 = new Object(); //ILLEGAL!
```

16.

```
/**
 Class for diamonds to be drawn on the screen. For this class, a diamond is completely
 determined by its diameter.
 (Screen character spacing determines the rest of the figure.)
 Inherits getOffset, setOffset, and drawAt from Figure.
*/
public class Diamond extends Figure
{
    private int diameter;

    public Diamond()
    {
        super();
        diameter = 0;
    }

    public Diamond(int theOffset, int theDiameter)
    {
        super(theOffset);
        diameter = theDiameter;
    }

    public void reset(int newOffset, int newDiameter)
    {
        setOffset(newOffset);
        diameter = newDiameter;
```

```
}

/**
 Draws the figure at the current line.
*/
public void drawHere()
{
    drawTop();
    drawBottom();
}

public void drawTop()
{
    int startOfLine = getOffset() + (diameter/2);
    spaces(startOfLine);
    System.out.println('*');
    int count;
    int lineCount = (diameter/2) - 1;
    int insideWidth = 1;
    for (count = 0; count < lineCount; count++)
    {
        startOfLine--;
        spaces(startOfLine);
        System.out.print('*');
        spaces(insideWidth);
        System.out.println('*');
        insideWidth = insideWidth + 2;
    }
}

public void drawBottom()
{
    int startOfLine = getOffset();
    int count;
    int lineCount = (diameter/2);
    int insideWidth = 2*lineCount - 1;
    for (count = 0; count < lineCount; count++)
    {
        spaces(startOfLine);
        System.out.print('*');
        spaces(insideWidth);
        System.out.println('*');
        insideWidth = insideWidth - 2;
        startOfLine++;
    }
    spaces(startOfLine);
    System.out.println('*');
}

private static void spaces(int number)
{
```

```
        <该定义和图7-12中的Triangle类的spaces方法相同。>
    }
}
```

这个问题指出了另一种定义此种字符图形类的好方法，那就是用一个辅助类，内置一些公有的静态方法，如spaces方法，还有其他方法，比如画水平线、大V型及倒置的大V型等。

17. 这样不合法，因为Figure（如图7-14所示）是一个抽象类。
18. 是的，这样合法。
19. *重写*是指对基类中的方法名重新定义，使之在派生类中有不同的定义。*重载*是指对一个方法名给出多个定义，通过不同的参数列表来区分。重载可以在任何类中进行，不管是否显式地给出了基类的名。区分二者的方法是：当*重写*方法定义时，在派生类中的新定义具有完全相同的参数数目和类型。另一方面，如果派生类中的方法有不同数目的参数，或者参数的类型不同，则派生类中就同时有这两个方法，那就成了*重载*。重载也不必涉及基类，只要在单独的一个类的定义中给出两个方法定义就可以。
20. 重写。
21. 重写。
22. 重载。
23. *多态*是指采用动态绑定使不同的对象对于同一个方法名有不同的行为。
24. *动态绑定*和*延迟绑定*是同一件事物的不同说法（因此我们只给出一种定义和一个例子）。动态绑定（或延迟绑定）是一种处理调用的方法可能在派生类中被重写的方式。有了动态绑定，方法调用的意义在运行程序之前，并不与其在程序中的位置绑定在一起。通俗一些说，当决定要使用哪一个重写的方法定义时，计算机实际使用的是（用new）创建对象时的类的定义，它并不一定是命名该对象的变量的类型中的定义。Java使用动态绑定。如图7-13所示。如果Java不使用动态绑定，那么两处对drawAt方法的调用都将使用图7-10中Figure类给出的drawHere方法的定义，这样在屏幕上出现的就只是两个星号。
25. 这个问题可能没有明确的答案。在*多态*的最初定义中，重载也被认为是多态的一个例子，有些书仍旧使用这个定义。在最新的用法中（包括本书在内），重载方法名*不是*多态的例子。
26. 大多数程序员把封装、继承及多态作为面向对象程序设计的主要特征。
27. 否。它们将产生不同的图形。前者画出一个盒子，后者画出一个三角形。
28. 否。你需要自行定义（或者从书中的文本中照搬，或者从网页上本书的源代码中复制）。
29. 没有不同之处。
30. 在从JFrame派生来的GUI的构造器中。

● 编程项目

1. 写一个名为Employee的类的定义，这个类的对象是雇员的记录。该类应当是图7-1中Person类的派生类。雇员记录中有姓名（从Person类继承得到）、用单个double类型的值表示的年薪、用单个int类型值表示的雇佣日期的年份，以及用String类型的值表示的社会安全号（SSN）。社会安全号原本可以保存为整数，不过由于可能会超过整数的范围，因此使用字符串（毕竟社会安全号只是任意的标识符，没有任何数字的性质）。别忘了这个类还需要补充一些构造器、访问方法及设置方法，还要一个equals方法。编写一个程序来完整地测试这个类的定义。
2. 定义图7-10中的Figure类的两个派生类，一个叫作RightArrow，另一个叫作LeftArrow。这两个类像Triangle和Box，但是它们将画出左向和右向的箭头，右向的箭头的形状如下所示。

```
                 *
                 * *
                 *   *
    ************       *
                 *   *
                 * *
                 *
```

箭头的大小由两个数字决定，一个是箭尾的长度，在前面的例子中是12，另一个是箭头底边的宽度，在此例中是7。底边的宽度不能是偶数，构造器和设置方法应当检查确保它是奇数。为每个类写一个测试程序来测试其中的各个方法。可以假设箭头底边的宽度是3或者更大。

3. 编写一个名为Doctor的类的定义，其对象是一个诊所的医生。该类是图7-1中Person类的派生类。Doctor记录中有医生的姓名（从Person类中继承）、专长（例如儿科Pediatrician、产科Obstetrician、普通General Practitioner等，因此用String类型），以及表示挂号费的double类型值。别忘了这个类还需要补充一些构造器、访问方法及equals方法。编写一个程序来完整地测试这些方法。
4. 编写名为Patient和Billing的两个类，其对象是诊所中的记录。Patient从图7-1中的Person类派生出来。Patient记录中有病人的姓名（从Person类中继承）、社会安全号（如编程项目1中所述，使用String类型）。Billing对象将包含一个Patient对象和一个Doctor对象（来自编程项目3）。这两个类还各自需要补充一些构造器、访问方法及equals方法。先编写一个驱动程序来测试所有的方法，然后写一个测试程序，创建至少两个病人，至少两个医生以及至少两条Billing 记录，并打印出全部Billing记录的总收入。
5. 创建一个名为Vehicle的基类，其中有制造商的名称（类型是String）、发动机气缸数（类型为int）以及车主（图7-1中的Person类型）。然后创建一个Truck类，从Vehicle派生，并添加下列属性：以吨计量的载重量（double类型，因为可能有小数部分）以及以吨计量的拖动能力（double类型）。这个类还需要补充一些构造器、访问方法及equals方法。编写一个驱动程序来测试这些方法。
6. 创建两个类，名为RightTriangle和Rectangle，它们都是图7-10中的Figure类的派生类。再定义Rectangle的派生类Square。这3个类除了继承的方法之外，都各有两个方法来计算面积和周长。重定义drawHere方法，以及补充一些构造器和访问方法。Square类应当只有一个边长side，并自动把长宽都设置为等于边长。尽管字符的宽和行间距不等，还是用它们作尺度单位，这样正方形看起来就不是正方形了（正如本章中所讨论的，Box看起来也不是正方形了）。注意Square类和Box产生同样的图形，但我们是通过不同的途径得到这两个类的。编写一个驱动程序来测试所有这些方法。
7. 创建一个新类名为Dog，令其为图5-20的PetRecord类的派生类。新类添加了breed（String类型）和boosterShot（boolean类型，若该宠物打过防疫针则为true，否则为false）属性。给新类补充合理的构造器和访问方法。编写一个驱动程序来测试这些方法，然后写一个程序，读入5个Dog类型的宠物，打印出所有超过2岁且没有打过防疫针的狗的名字和品种。
8. （本项目需要阅读7.4节）在下面一些方面改进图7-18所示的ButtonDemo GUI类。当GUI刚开始显示时，两个按钮都可见，但是当一个按钮按下时，这个按钮就要消失，只有另一个按钮可见。此后仅有一个按钮可见，当这个按钮按下，它就消失而另一个按钮重新出现。当某个按钮按下时，其颜色的变化同图7-18所示相同，并且显示代表“Sunny”的兴高采烈的笑脸或者显示代表“Cloudy”的愁眉苦脸（另一个也相应消失）。刚开始运行时，GUI中并不显示任何一张脸。等按下按钮10次之后（两个按钮的合计次数），显示一条消息说：“只能再按一次了”。再按一次按钮之后，程序结束。